

R Programming In Statistics



Prof Dr Balasubramanian Thiagarajan

ISBN: 978-93-5737-722-5

Preface

Every professional needs to perform statistical analysis in some form or the other. In order to perform this task various software tools are available. Majority of them are paid software. R programming which is an open source tool can be used to perform statistical analysis. Since it is an open source tool many front end GUI's are available to make the job easier for the user. In this book the most popular GUI RStudio is used. RStudio is a most powerful GUI front end for R programming which has been designed to use all the features of this language with ease. This book has been authored with a novice user in mind. Various steps in statistical analysis have been explained in detail using a large number of screenshots. Codes used have been clearly illustrated. The book has been structured in such a manner to ensure that basic concepts have been clearly explained with the help of screenshots before taking on challenging analytical problems.

Towards the end of the book the reader is provided with an additional resource which gives out all the codes used in this book as well as those additional ones that have not found their place in the book. Learning R coding is not difficult provided the reader spends time practicing the same. The reader is encouraged to execute all the codes provided in the R_code manual which has been provided at the end of the book. R programming can be compared to that of SPSS (the popular statistical analytical tool) as far as its ability to perform statistical analysis. One tip the author wishes to provide to the reader who is attempting to make data entry within the RStudio environment. It is always better to import data into RStudio for performing data analysis. Data can be imported from Excel , google spread sheets etc.

The reader is encouraged to download the install the software and libraries that have been described in the book and to try them out.

Advantages of R Programming :

1. It is a powerful statistical tool
2. It is open source and hence it is free
3. It is an excellent tool that can be used to perform visual analysis of a dataset. It can create different types of charts and graphs, thereby facilitating accurate analysis of data.

Being the first edition author invites comments from the readers. The same be mailed to:

[Email](#)

About the Author



Prof Dr Balasubramanian Thiagarajan is a practicing otolaryngologist
Former professor and head Department of otolaryngology
Government Stanley Medical College
Chennai
Former Registrar
The Tamilnadu Dr MGR Medical Univeristy
Guindy
Chennai.

Currently

Dean

Sri Lalithambigai Medical College
Maduravoil
Chennai

Contents

Introduction 7

Unique features of R programming: 10

Installation R base software: 10

Installation of RStudio: 18

Why a programming language like R should be learnt by a non-programmer? 23

RStudio ideal settings & RGui 24

Updating R and RStudio: 28

RGui: (R Base software) 31

Print: 36

GUI Preferences: 39

View menu: 40

Packages menu: 43

Windows Menu : 48

Help Menu: 50

Getting started: 54

R-Studio 54

Console: 56

Types of Data in R 79

Data An Introduction 79

Operators in R Programming 140

Assignment Operators: 164

These operators are used to assign values to vectors. 164

Left assignment: 164

<- 164

= 164

<<- 164

These operators can be used interchangeably. 164

c indicates concatenate in R language. 164

Miscellaneous operators: 167

Statistical summary function:	169
Simulation and statistical distributions:	171
Functions in R Programming	177
List function:	203
Data Entry in R Programming	233
Data Analysis in R Programming	255
Exploratory data analysis:	263
Measures of central tendency:	267
One Sample T-Testing:	283
Hypothesis Testing in R Programming	283
Two Sample T-Testing:	285
Directional Hypothesis:	287
One Sample Mu test:	288
Bootstrapping in R Programming:	291
Time series analysis using R:	294
Tidyverse	299
Anova	320
Post-hoc tests in R:	333
Descriptive Statistics	335
Mean:	341
Median:	343
Interquartile range:	344
Standard deviation and variance:	344
Summary:	347
Coefficient of variation:	347
Mode:	347
Correlation:	351
Mosaic plot:	353
Bar plot:	353
Histogram:	355

Box plot: 357

Dot plot: 357

Scatter plot: 357

Exploratory Data Analysis 359

Regression Analysis using R 364

Pie chart: 373

R Charts and Graphs 373

Bar plot: 377

Boxplots: 382

Line graphs using R: 389

R Scatterplots: 395

Creating the scatterplot: 396

Introduction

R is a language and environment for statistical and graphics. This GNU project is similar to the “S” language and environment that was developed by Bell laboratories. Even though R can be considered as a different implementation of S, there are some important differences. Most of the code written for S runs unaltered under R.

In 1992, Ross Ihaka and Robert Gentleman created R at the University of Auckland. This was to enable the students to use this as a statistical tool. Initial version was released in 1995. Currently it is being maintained by the R Development Core Team.

R provides a variety of statistical (linear and non-linear modelling, classical statistical tests, time series analysis, classification, clustering etc). It also provides graphical techniques and is highly extensible.

One major strength of R is the ease with which well-designed publication quality plots can be produced, including mathematical symbols and formulae when needed.

1. It is a free and open source tool.
2. It has a large community of users
3. It is an independent platform and can be run without a compiler.
4. Can be considered to be the Gateway for lucrative career
5. Has a robust visualization library - R comprises libraries like ggplot2, plotly that offer aesthetic graphical plots to its users. R is recognized for its stunning visualizations which gives it an edge over Data science programming languages.
6. Used in almost every Industry
7. Distributed computing - In distributive computing, tasks are split between multiple processing nodes to reduce processing time and to increase efficiency. R has packages like ddr and multiDplyr that enable it to use distributed computing to process large data sets.
8. Interfacing with Databases - R contains several packages that enable it to interact with databases like ROracle, Open database connectivity Protocol, Rmy SQL, etc.
9. Data Variety - R can handle a variety of structured as well as unstructured data. It also provides various data modeling and data operation facilities due to its interaction with databases.
10. Compatible with other programming languages - Most of the functions are written in R itself, C, C++ or Fortran can be used for computationally heavy tasks. Java, .NET, Python can also be used to manipulate objects directly.

R code can be run without any compiler. It is an interpreted language and hence compiler is not need to run the code. Calculations are done with vectors. R is actually a vector language, hence anyone can add functions to a single vector without putting in a loop. R is hence powerful and faster than other languages.

Feature of R include:

1. Data inputs and data management. Data inputs such as data type, importing data and keyboard typing.
2. Data management such as data variables, operators.

Pros of R language:

1. It is the most comprehensive statistical analysis package, and new ideas often appear first in R.
2. R is an open source and can be run anywhere any time.
3. It is cross platform and runs on many operating systems.

Cons of R language:

1. The quality of some packages in R is less than perfect.
2. There is no customer support of R language.

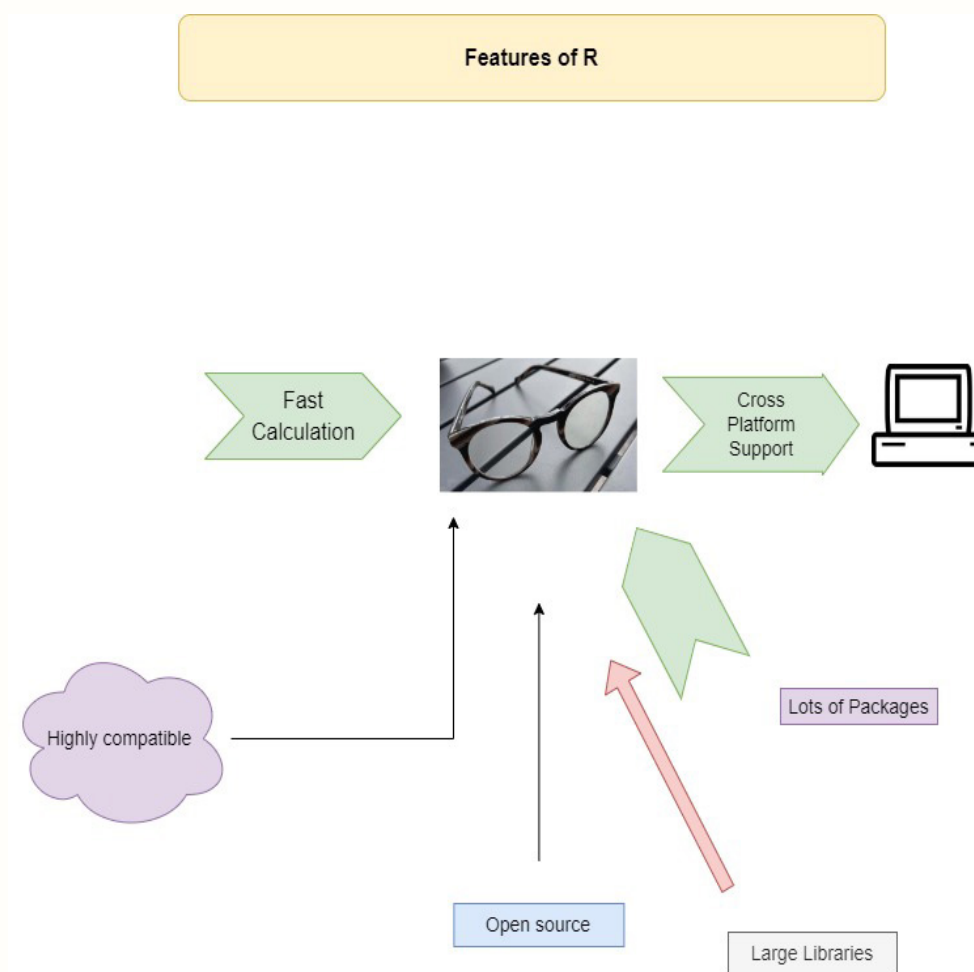
The R Environment:

This is an integrated suite of software that can be used for data manipulation, calculation and graphical display. It includes:

1. An effective data handling and storage facility
2. A suite of operators for calculations on arrays, in particular matrices
3. A large, coherent, integrated collection of intermediate tools for data analysis
4. Graphical facilities for data analysis and display either on-screen or on hard copy
5. A well developed, simple and effective programming language which includes conditions, loops, user defined recursive functions and input and output facilities.

The term environment is intended to characterize it as a fully planned and coherent system rather than an incremental accretion of very specific inflexible tools.

R has been designed around a true computer language, and it allows users to add additional functionality by defining new functions. R also has its own LaTeX like document format which is used to supply comprehensive documentation both on-line in a number of formats and in hard copy.



Prerequisites before learning R:

Before one jumps into R, it is highly recommended that they possess some basic knowledge of a few topics. These include:

1. Basic understanding of statistics, mathematics, and probability.
2. General understanding of data science and the process involved.
3. Basic understanding of various types of graphs and data representation techniques.

Unique features of R programming:

Since there are a large number of packages available, there are many handy features in R. They include:

1. Its ability to perform directly on vectors and hence does not require too much looping.
2. It can pull data from APIs, servers, SPSS files and many other formats.
3. It is very useful for web scraping.
4. It can perform multiple complex mathematical operations with a single command.
5. It can create attractive reports combined with plain text with code and visualizations of the results if R markdown feature is used.
6. Since the user base is large, new ideas and technologies appear in the R community first.

Installation R base software:

Step I : R Base needs to be installed first. R is maintained by an international team of developers and the software is available in multiple languages in their webpage “The Comprehensive R Archive Network”. From here the version appropriate to the User’s operating system can be downloaded. R is available for:

Windows operating system

Mac OS

Various flavors of linux

Installing R in windows is fairly simple as it comes bundled with its own installer which takes care of the entire installation process. As the user has to do is to double click on the downloaded binary file.

Step II: The windows executable file after being downloaded is double clicked to begin the installation process. All the user has got to do is keep clicking the next button till the confirmation screen appears saying that the process of installation is over. If the user is using a computer that is shared by others then Install for all users radio button needs to be selected to make the software available to all the users using the system. The first screen allows the user to choose the language of installation. R software is available in various common languages. It is preferable to allow the installation into the default folder created by the installer than customizing the process of installation. Since the user will have to install an Integrated Development Environment (IDE) software after installing R base software it will be fairly straight forward for the IDE to use R base software as it has been installed in to the default folder

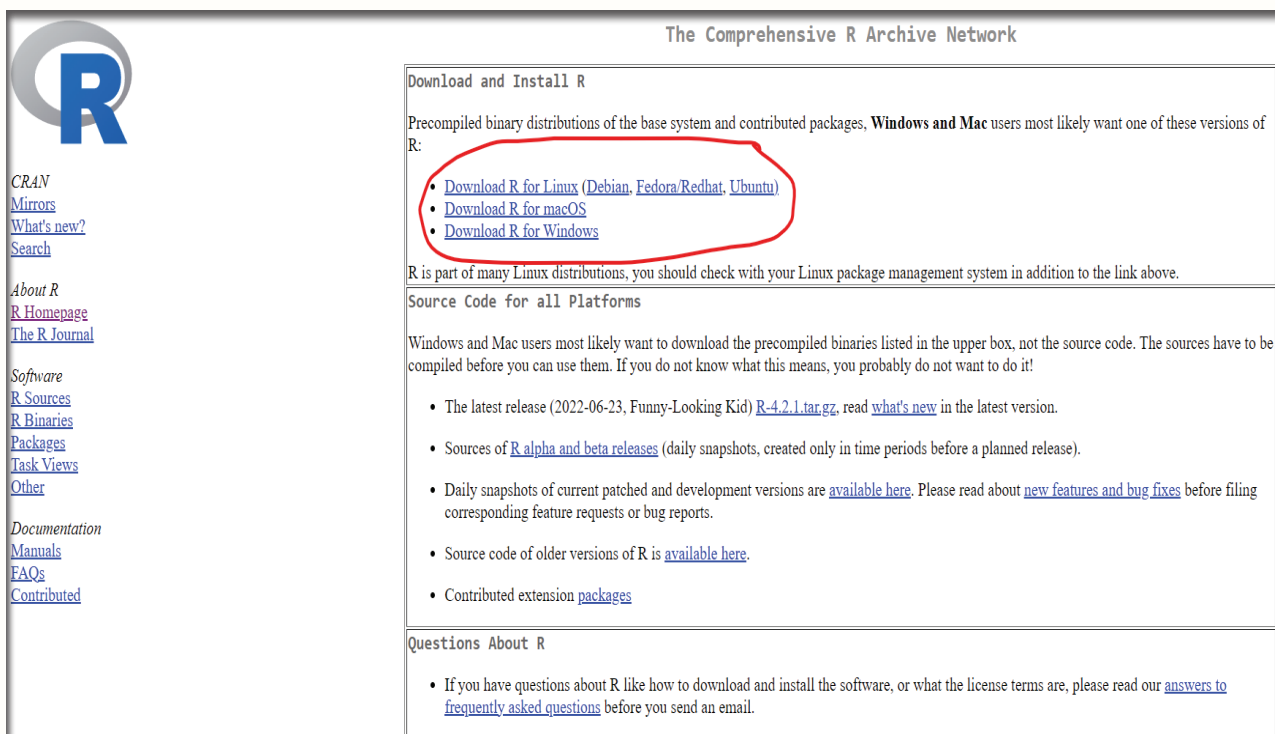
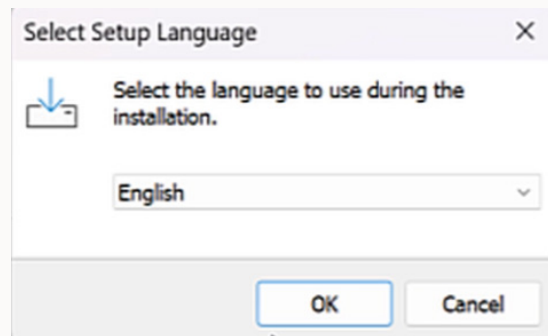


Image showing CRAN webpage where the various flavors of R are available for download



Image showing the official R project webpage



In the first screen shown above the language of the installation needs to be chosen before clicking on the OK button

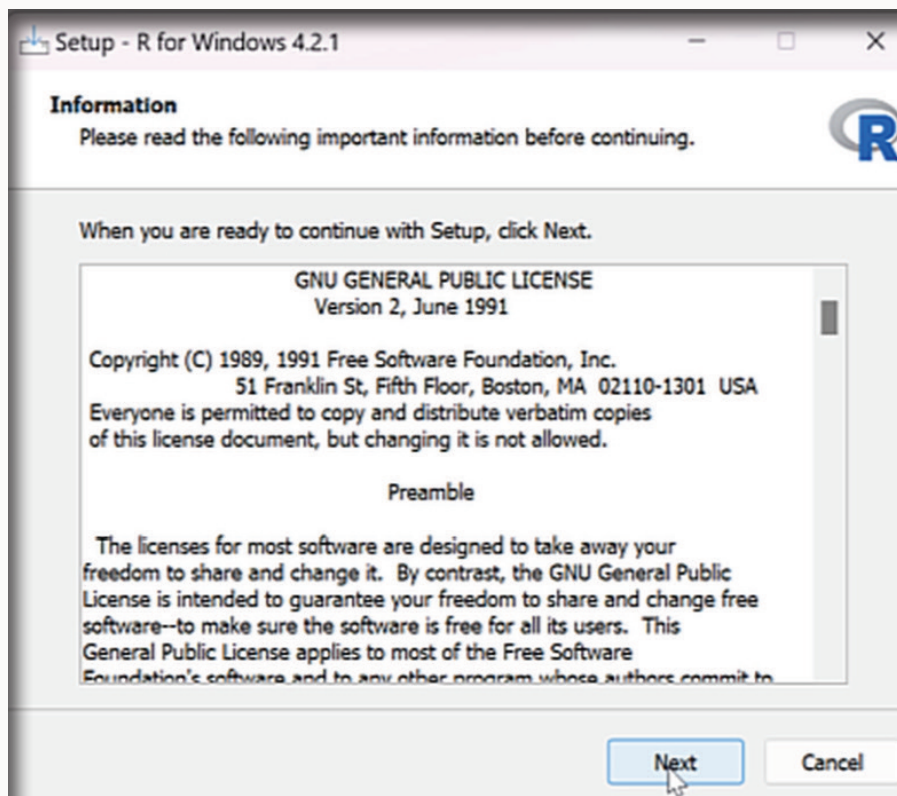


Image showing GNU licence screen which needs to be accepted by clicking the next button

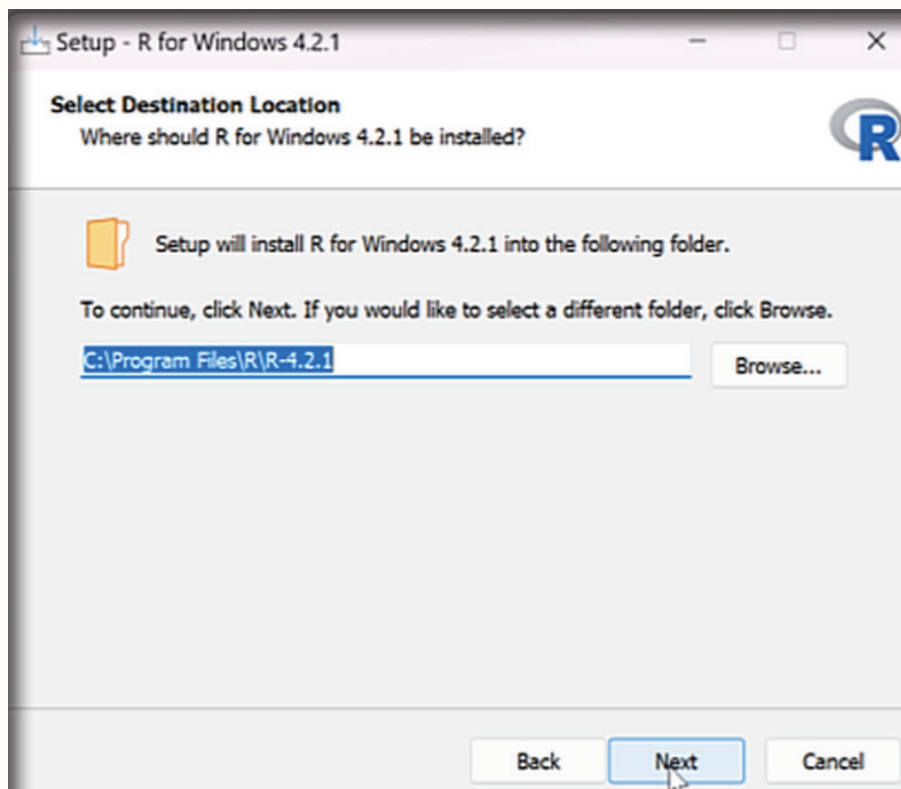


Image showing the screen that gives the choice of destination of location to the user. It is ideal for the user to allow the default settings by clicking on the next button. If the system has an SSD disk installed then installation is preferred in that disk as it would speed up the application process. If the user's system has multiple hard disks and one of them happens to be a SSD it is preferable to install it there.

R comes with both 32 bit AND 64 bit versions. The user will have a dilemma in choosing which version to use. Actually it does not matter as both versions use 32-bit integers, which indicates that they compute numbers to the same numerical precision. The difference occurs in the way each version manages the system memory. 64-bit R uses 64-bit memory pointers and 32-bit uses 32-bit memory pointers, this means that 64-bit has a larger memory space to use.

It should be pointed out that 32-bit builds of R are slightly faster than 64-bit builds. On the flip side 64-bit builds can handle larger files and data sets with fewer memory management problems. Hence if the operating system does not support 64-bit programs, or the installed RAM is less than 4 GB then it is ideal to install 32-bit R software. If the system supports 64-bit then the installer would install both versions of R.

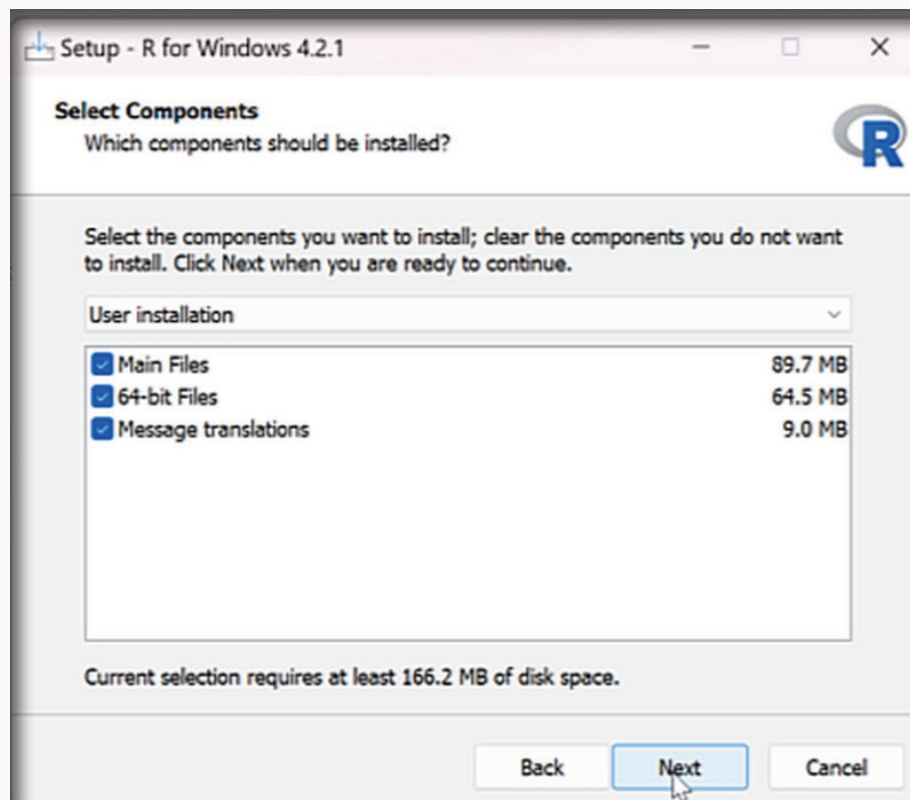


Image showing the screen that prompts user to select the desired components for installation. The user should choose the Main Files, 64-bit files if desired and Message translations if needed. The default settings is preferred and advisable. If the user wants 32 bit installation only, then 64-bit Files can be unchecked.

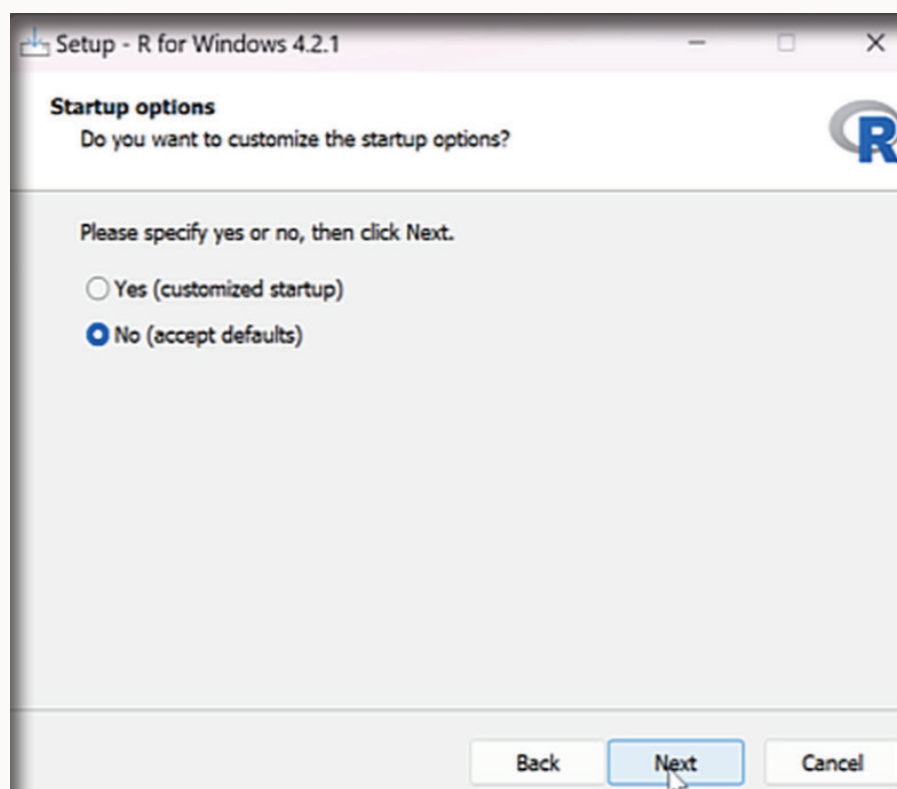


Image showing startup options window

Startup options:

When R is started, it will by default source a .Rprofile file if it exists. This allows the user to automatically tweak the R settings to meet the everyday needs. The startup package extends the default R startup process by allowing the user to put multiple startup scripts in a common "Rprofile.d" directory. If customization is needed for startup then during installation "customize startup radio button is selected" and in the ensuing window the customized file is pointed to enable customized startup. The user can have one file to configure the default CRAN repository and another one to configure their personal devtools settings. The user can also use a "Renviron.d" directory with multiple files defining different environmental variables like language etc,. One file could contain the private GITHUB_pat key.

This customization is needed for advanced users who are well versed in R language scripting and advanced computing techniques. This step is narrated not to daunt the first time user but to illustrate the extensive customizations that are available within R environment which can be used if desired.

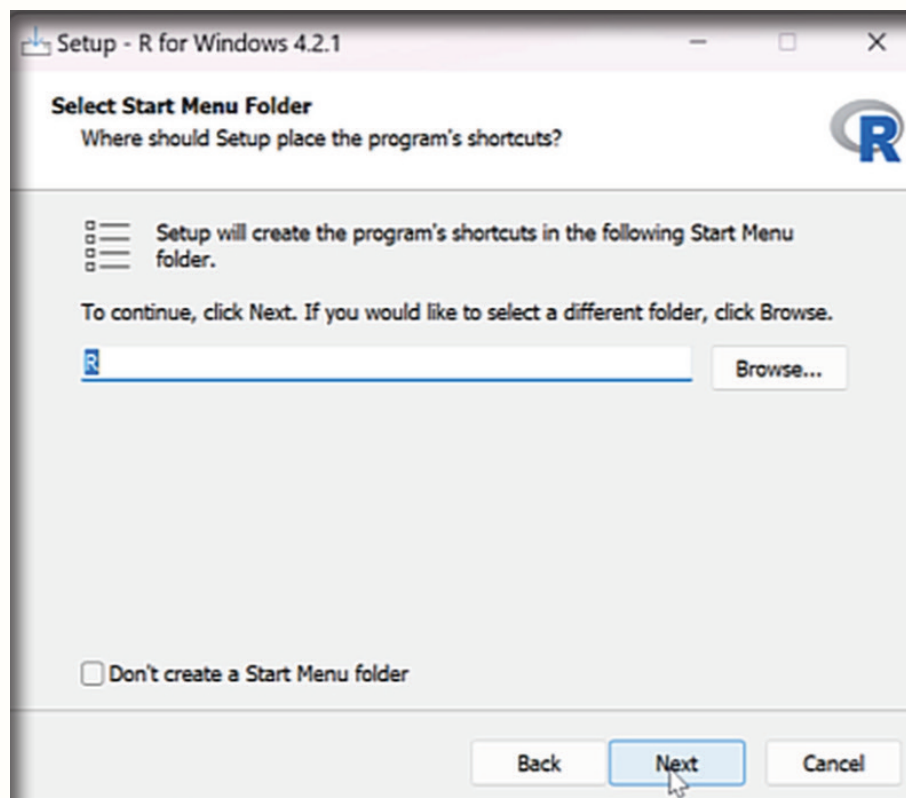


Image showing the prompt screen that allows the user to select the start menu folder where R short-cut is going to be stored. Here if the next button is clicked the default folder named R will be created in startup menu folder.

A small tip regarding the choice of installation folder in R programming installation:

If the user desires to install this software in a company owned computer where usually C drive access is not provided to the user as part of the company policy it is important to change the installation drive to where the user has access to. Installation will not progress if the user does not have access to the drive where installation folder is being created.

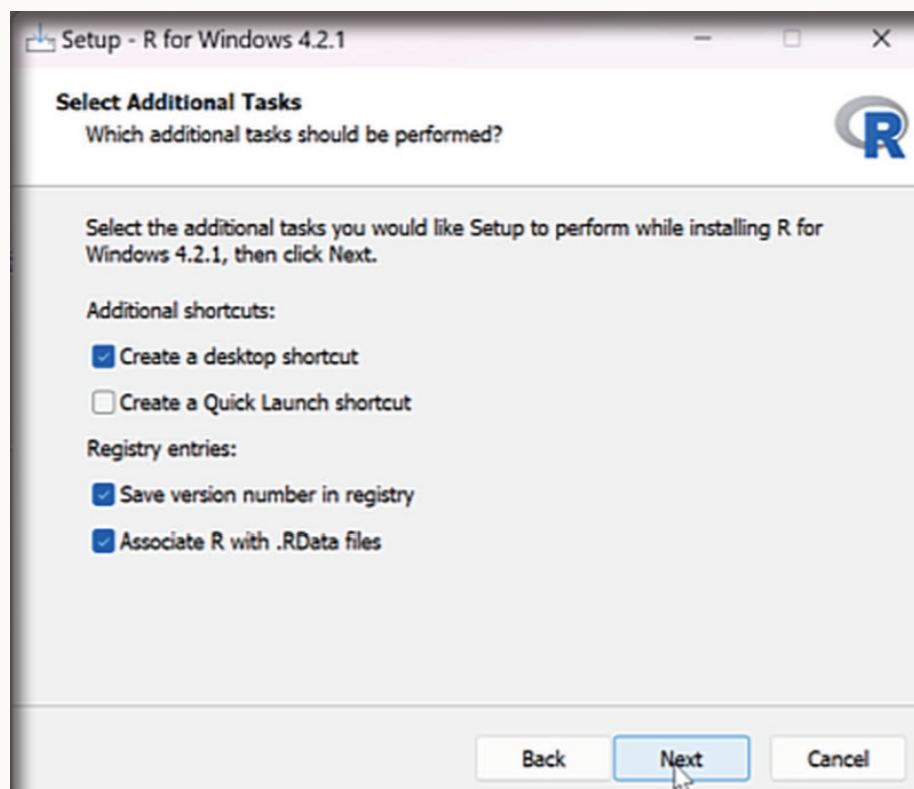


Image showing the installation screen where additional tasks can be selected during installation process.

In the image shown above the additional tasks that needs to be performed has been selected by default. The additional tasks already selected by default is sufficient for the installation to proceed. If the user desires to create a quick launch short cut then that box needs to be checked. Save version number in the registry helps in the process of identification of updates released if any. Another setting that has been chosen by default is Associate R with .RData files. This setting which is chosen by default will ensure that R files are associated with this software.

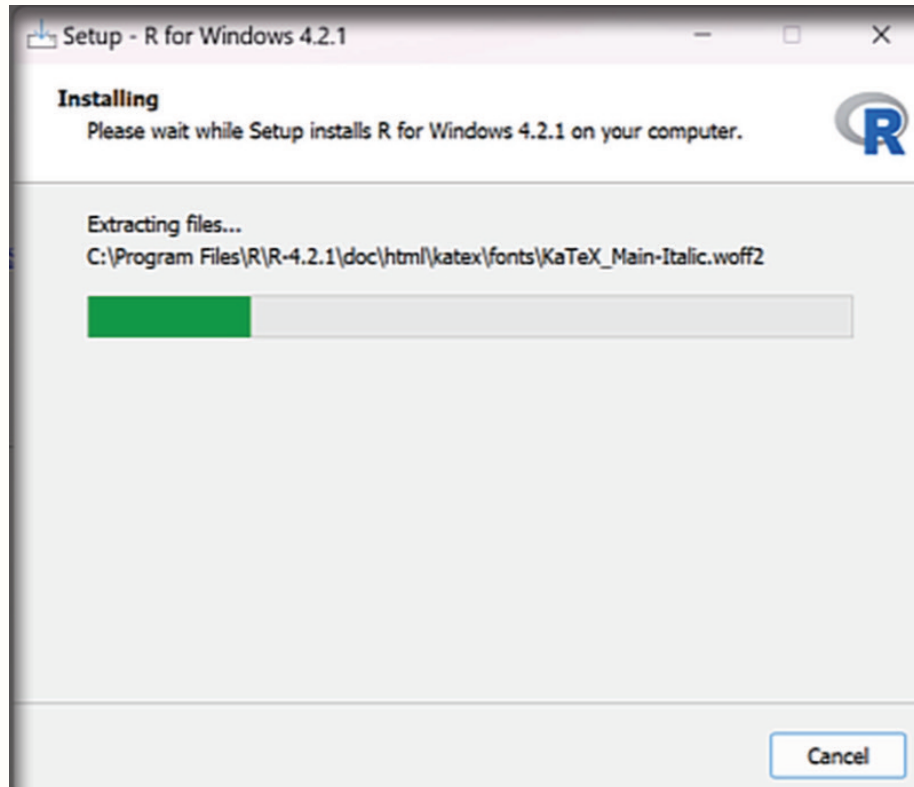


Image showing the file extraction process progressing

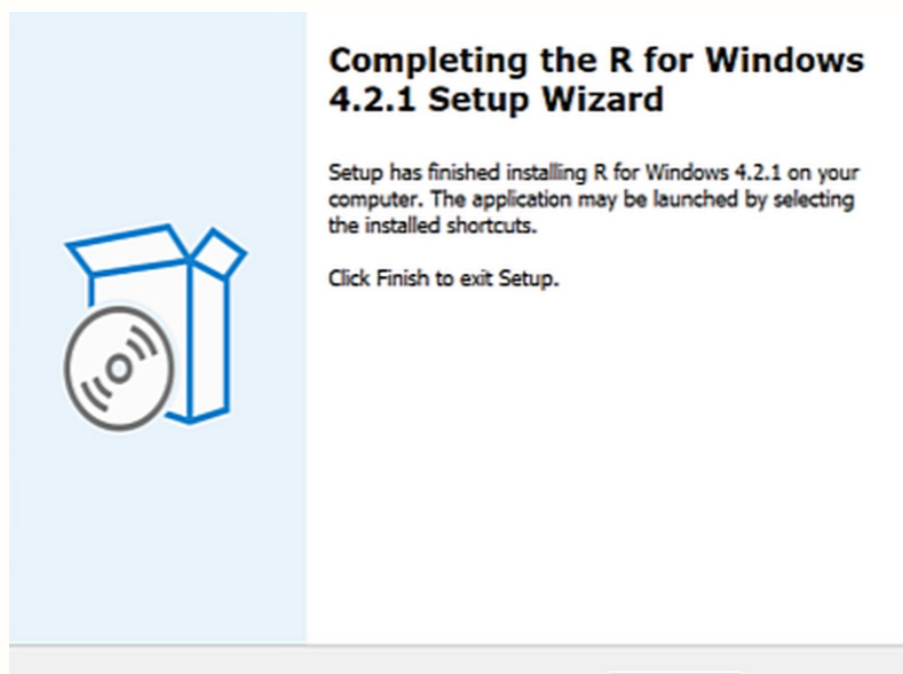


Image showing confirmation screen showing installation has been completed successfully

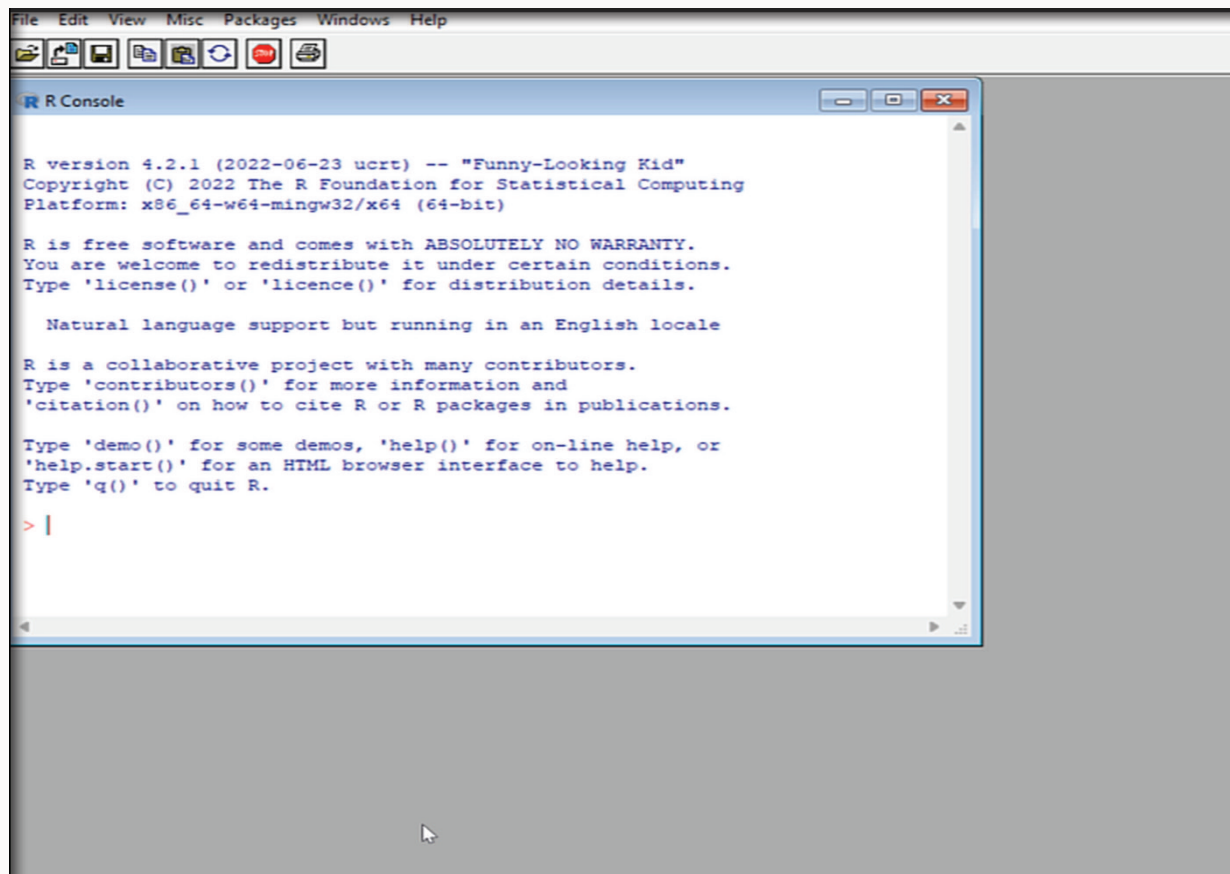


Image showing the R interface

Installation of RStudio:

RStudio is one of the most popular IDE (Integrated Development Environment) for working with R programming language. R studio should be installed only after installation of R base software. This would serve as a front end of R programming language.

Advantages of RStudio:

There are multiple ways to interface with R. Some common interfaces are the basic R GUI, R Commander and RStudio. Among these front end software for R programming language RStudio happens to be the best.

RStudio is designed to make it easy to write scripts. As soon as a new script is created, the windows within RStudio session adjusts automatically so that the user would be able to see both the script and the results in the console when the syntax is run. It has also the ability to call up potential syntax options while keying the scripts just by using the tab key.

RStudio makes it convenient to view and interact with the objects stored in the environment.

RStudio makes it easy to set the working directory and access files on the computer. This is more so true while working on windows environment. Without RStudio setting the working directory is the most tedious process in windows environment. Using RStudio one can navigate to folders on the computer in the “Files” window, view any files that are available in that folder, and set that folder as the working directory.

RStudio makes graphics much more accessible to a casual user. With the basic R programming one has to go to some lengths to save graphs, but with RStudio it has a window that makes the job simple.



Image showing the web page from which RStudio can be downloaded.

One of the easiest ways to reach this web page is to perform a google search for the term R Studio. It will take the user to the R studio page. In the RStudio web page free version of the software is chosen for the download. After the download is complete it can be executed for the installation process to continue.

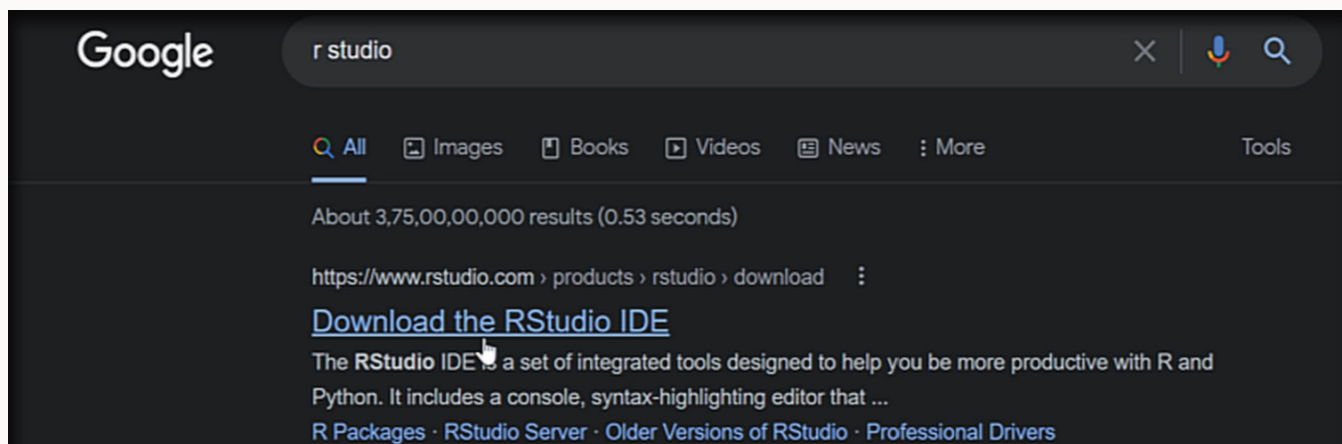


Image showing the google search result for RStudio

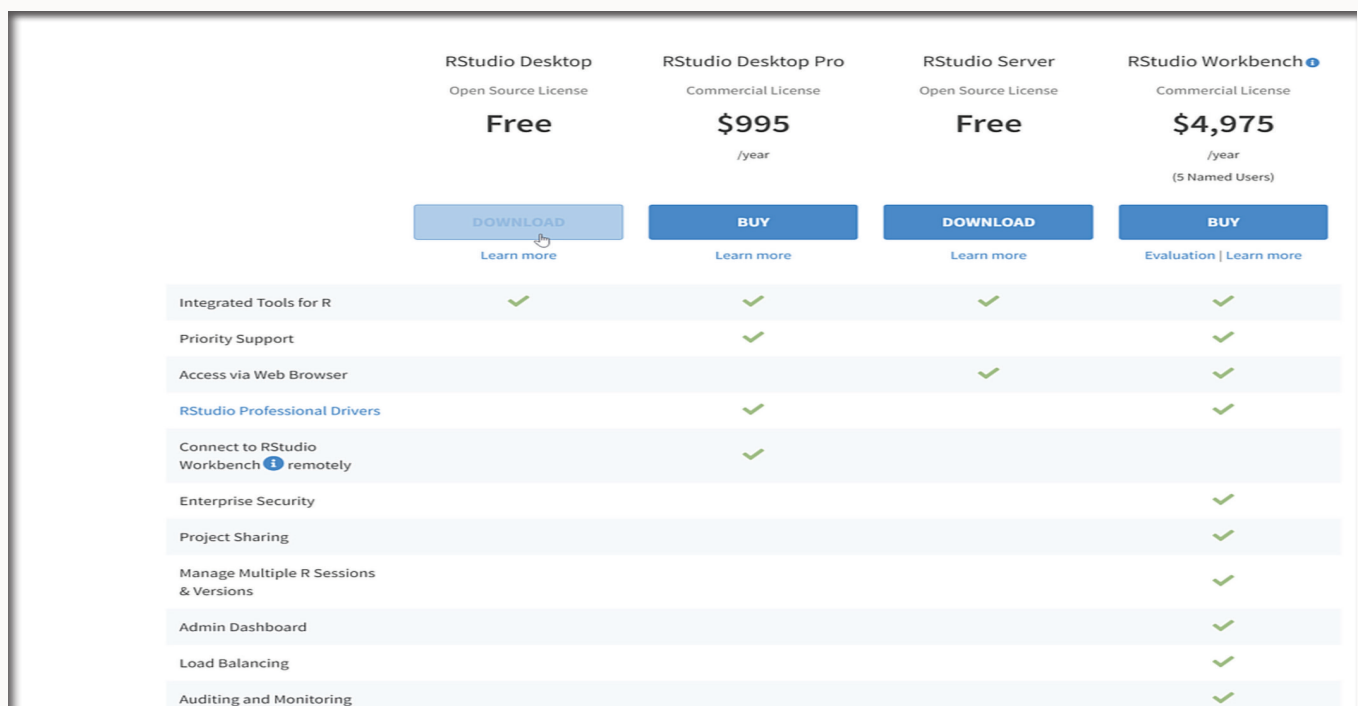


Image showing the download page for RStudio. RStudio Desktop Free version is chosen for download

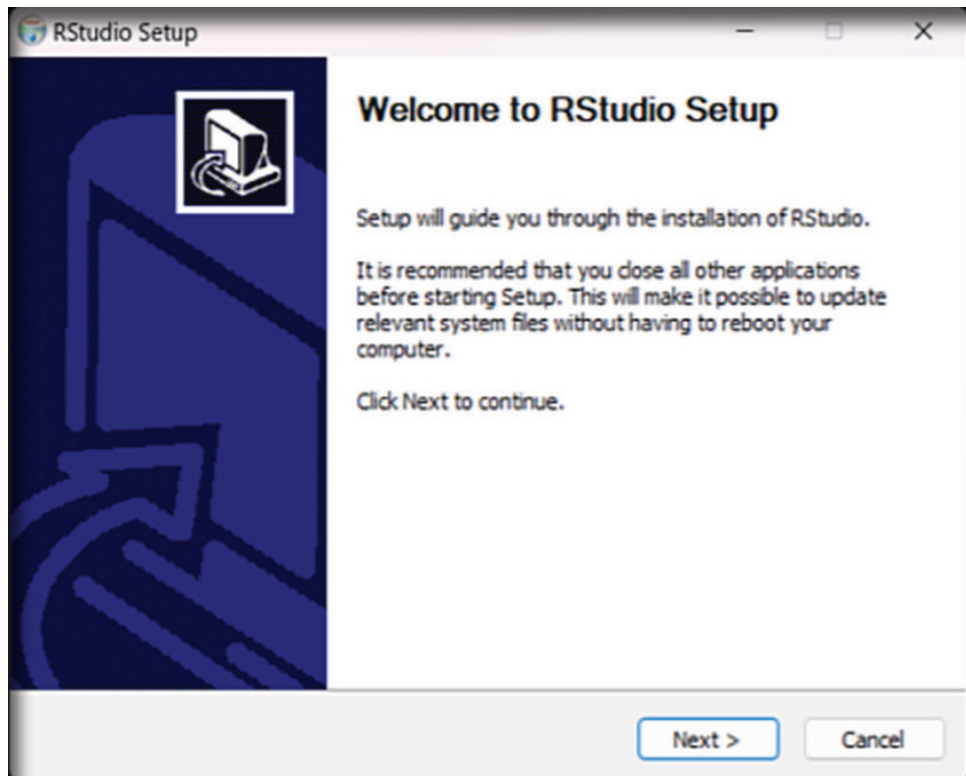


Image showing the RStudio setup screen.

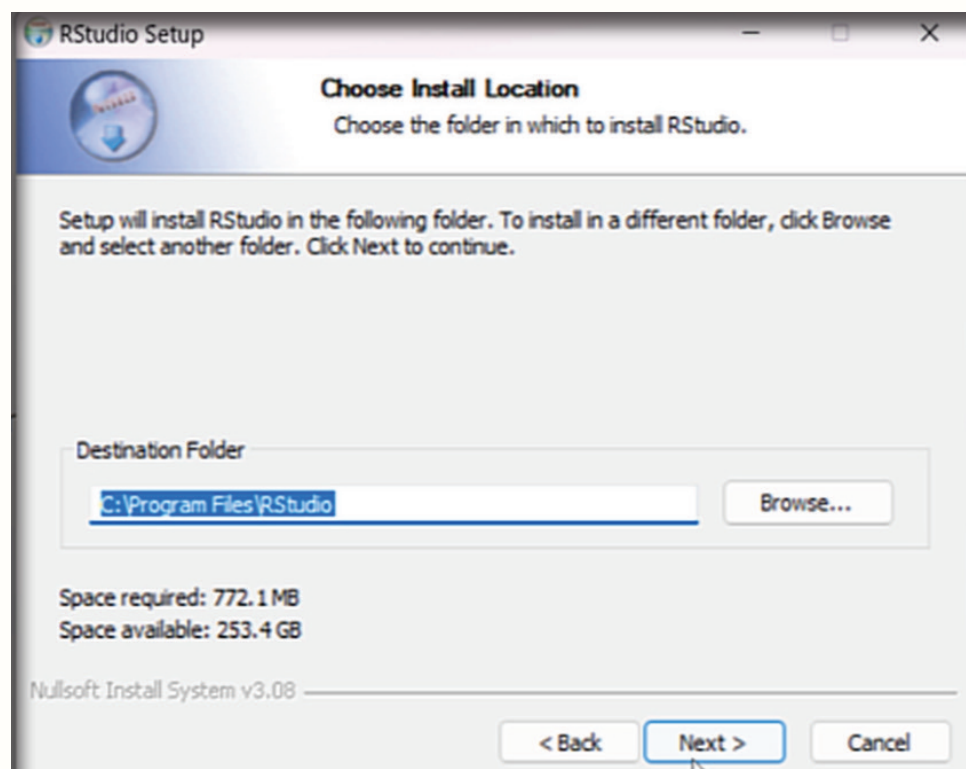


Image showing the screen where installation location can be chosen

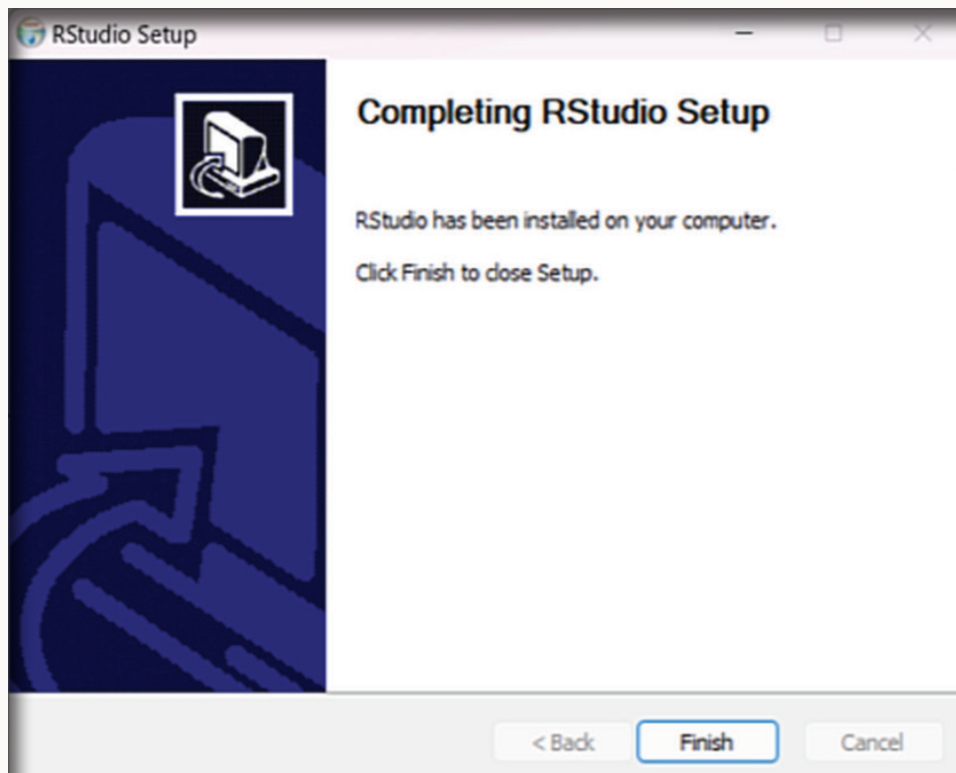


Image showing RStudio setup completed screen

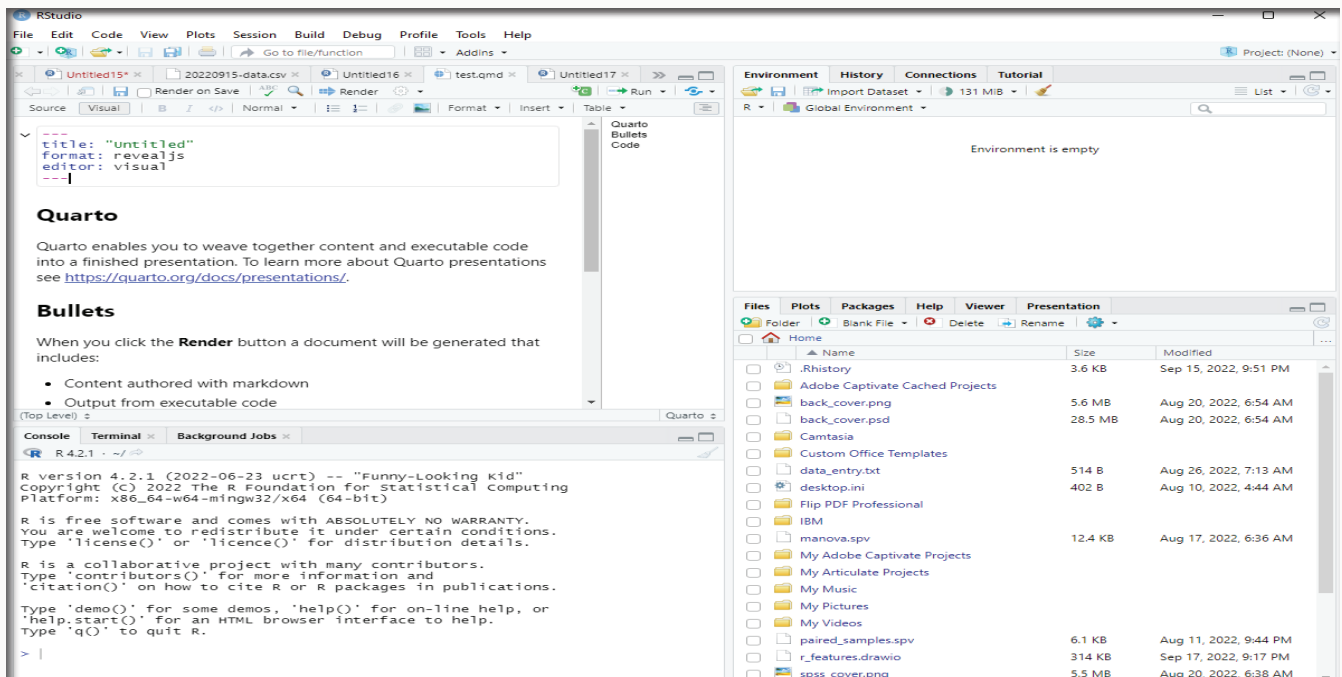


Image showing RStudio window

Why a programming language like R should be learnt by a non-programmer?

It must be stressed that R is a powerful programming language. It is used for a lot of quantitative data analysis, it has grown over the years to become a really powerful tool that specializes in handling data and performing customized computations with quantitative and qualitative data.

R language can be used to perform:

Statistical analysis

Corpus analysis

Development of online dashboards

Connection to social media APIs for data collection

Creation of reporting systems to provide individualized feedback to research participants.

Writing research articles, books and blog posts.

Learning new tools to analyze data is always essential. Theories change over time, and new insights into certain social phenomena are published every day. Knowledge might get outdated quite quickly. It should be pointed out that analytical techniques like mean, median, mode, quartiles, standard deviation etc., have remained the same. Programming languages allows the user to look at the data from a different angle.

RStudio ideal settings & RGui

For the first time user it is always better to adjust the following settings so that life for a programmer becomes that much easier. These settings are listed under Tools / Global options. Global options can be invoked by clicking on Tools button and selecting Global options from the drop down menu.

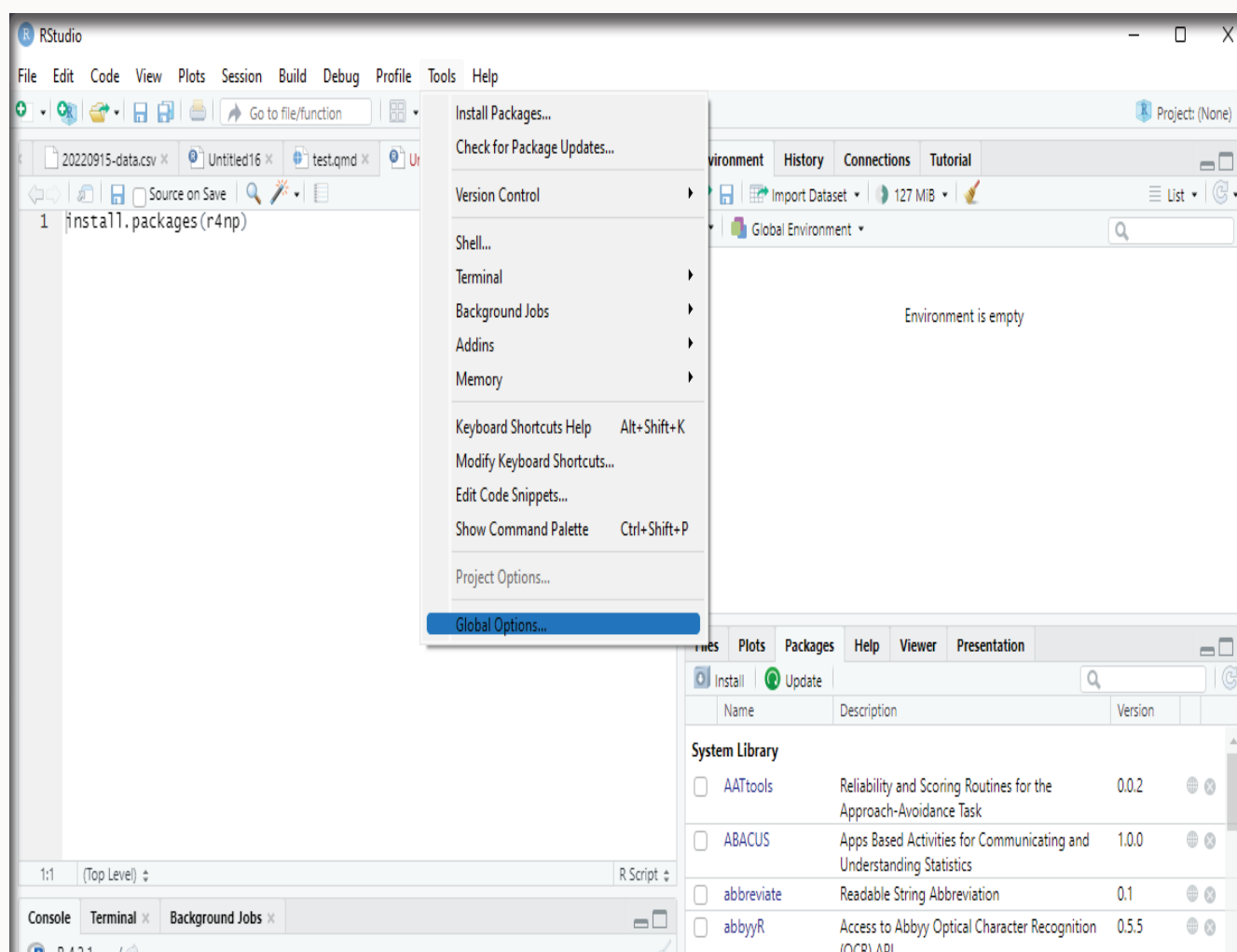


Image showing Global options listed under Tools menu

The following changes to Global options are recommended:

1. In the first tab (General > Basic) one should make one of the most significant changes. All options that starts with “Restore” should be deactivated. This will ensure that every time the user starts RStudio, it begins with a clean slate. It would seem counter-intuitive not to restart everything from where the user has left off, but is essential to make all the projects easily reproducible. Disabling this feature would also make it easy for collaborative work. The settings that need to be unchecked include:

- Restore most recently opened project at startup.
- Restore previously open source documents at startup.
- Restore .Rdata into workspace at startup.

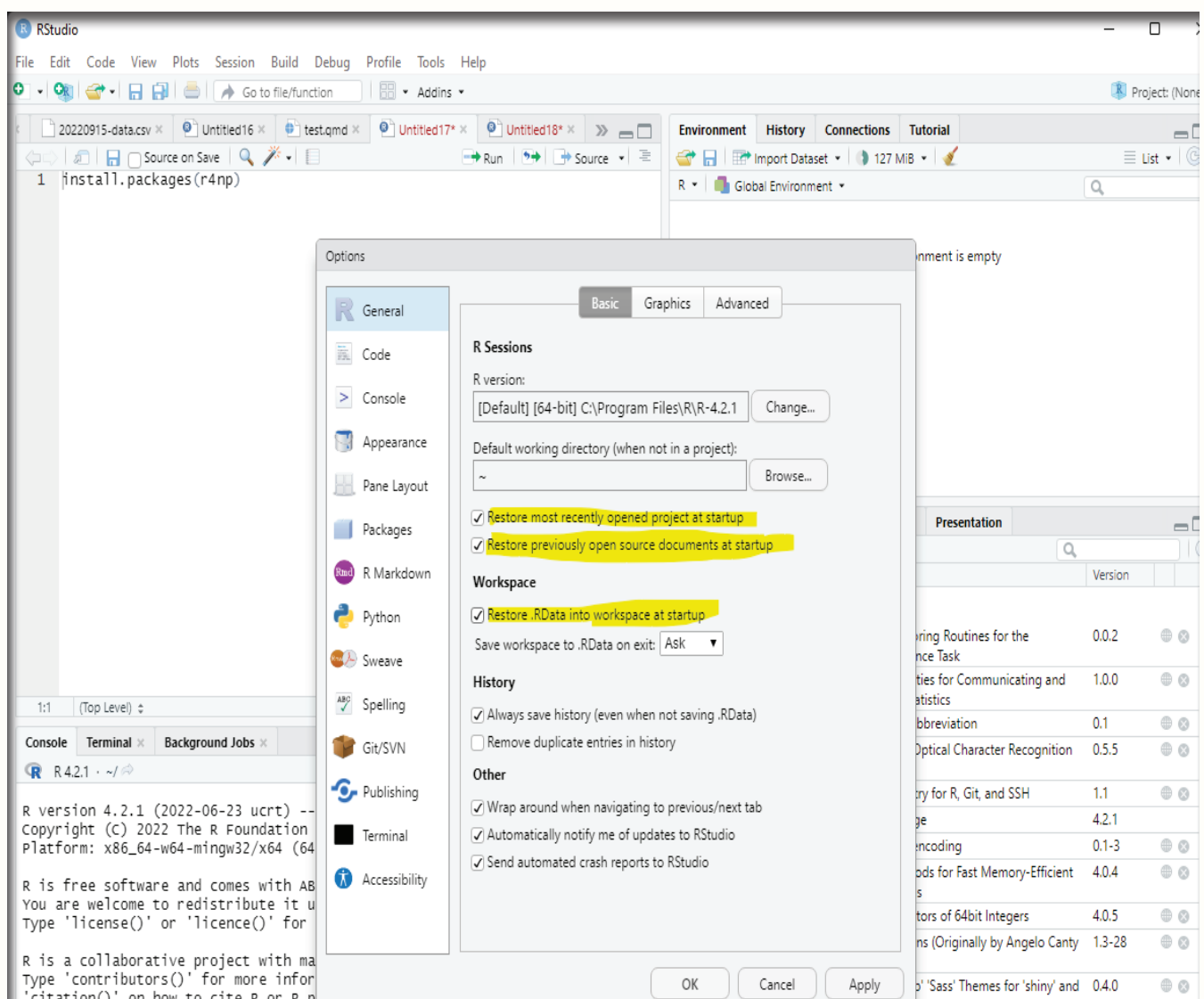


Image showing the Basic tab under General options. Note the highlighted settings needs to be unchecked. RStudio will restart to carry out the desired changes.

2. In the same tab under workspace, *Never* is selected for the setting *Save workspace to .RData on exit*. One could think that it is wise to keep intermediary results stored from one R session to another. Unchecking this setting would avoid future headaches.

3. In the Code > Editing tab it is made sure that at least the first five options are ticked. Especially the *Auto-indent code after paste*. This setting will save time when the user tries to format the coding appropriately, making it easier to read and comprehend. Indentation is the primary way of making the code look more readable and less like a series of random characters.

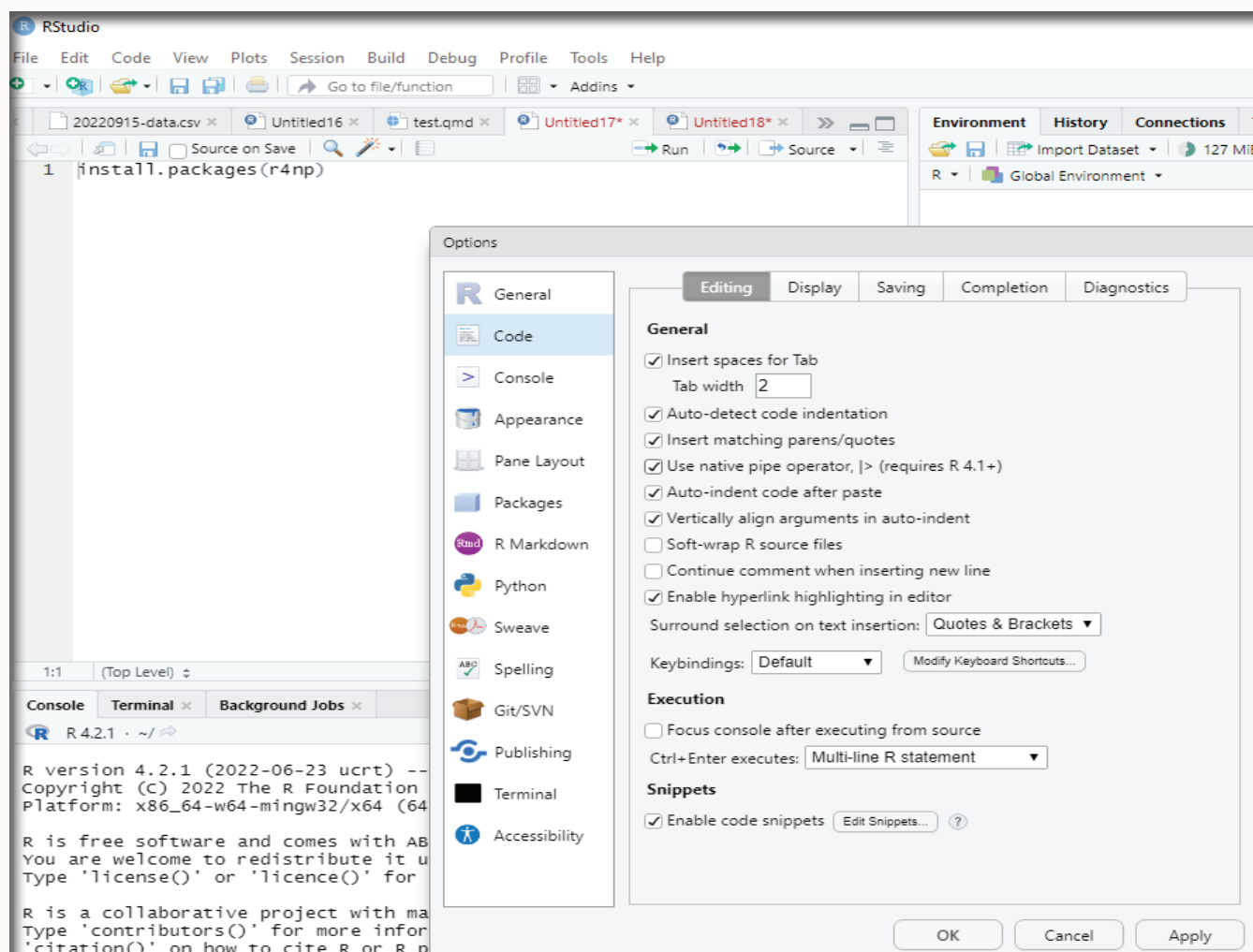


Image showing ideal Code settings that are preferred by the author.

At this point it should be stressed that there is no such thing as ideal settings. Settings are nothing but personal preference of the user. The fact that these settings are available ensures certain amount of flexibility to the user to manipulate. Individual users should be encouraged to play around with these settings and settle down with the most comfortable ones for their use. These are nothing but recommendations for the novice user.

4. In the Display tab under Code menu the first three options should be selected. Among these settings one particular setting is rather useful i.e., *Highlight selected line*. This is rather helpful in analyzing more complicated code, as it is helpful to see where the cursor is. One can also customize the workspace still further. The visually most impactful way to alter the default appearance of RStudio is to select *Appearance setting* and pick a completely different theme. There are no absolute right and wrong here. It is purely personal preference of the user.

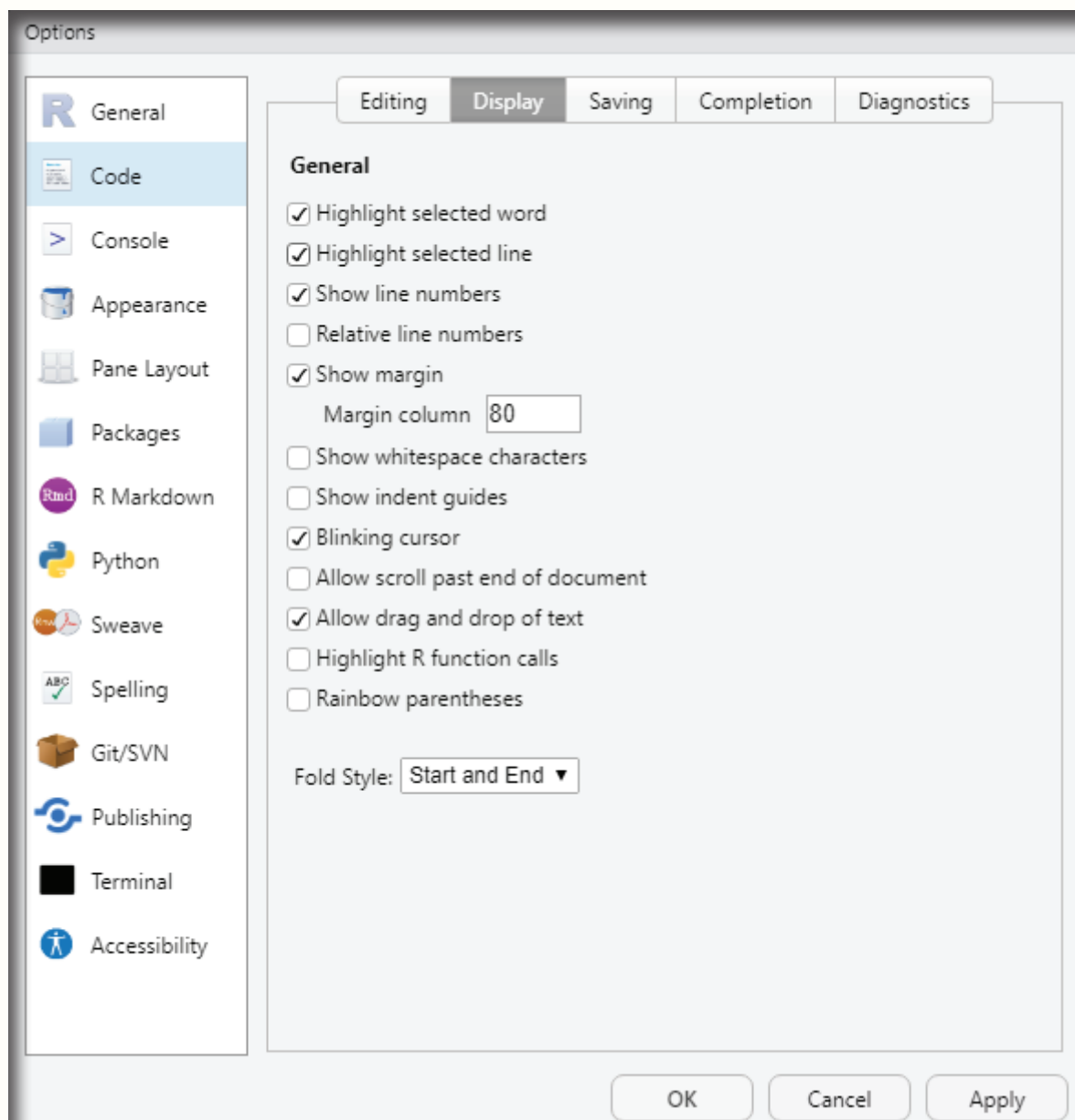


Image showing ideal Display settings chosen under Code menu

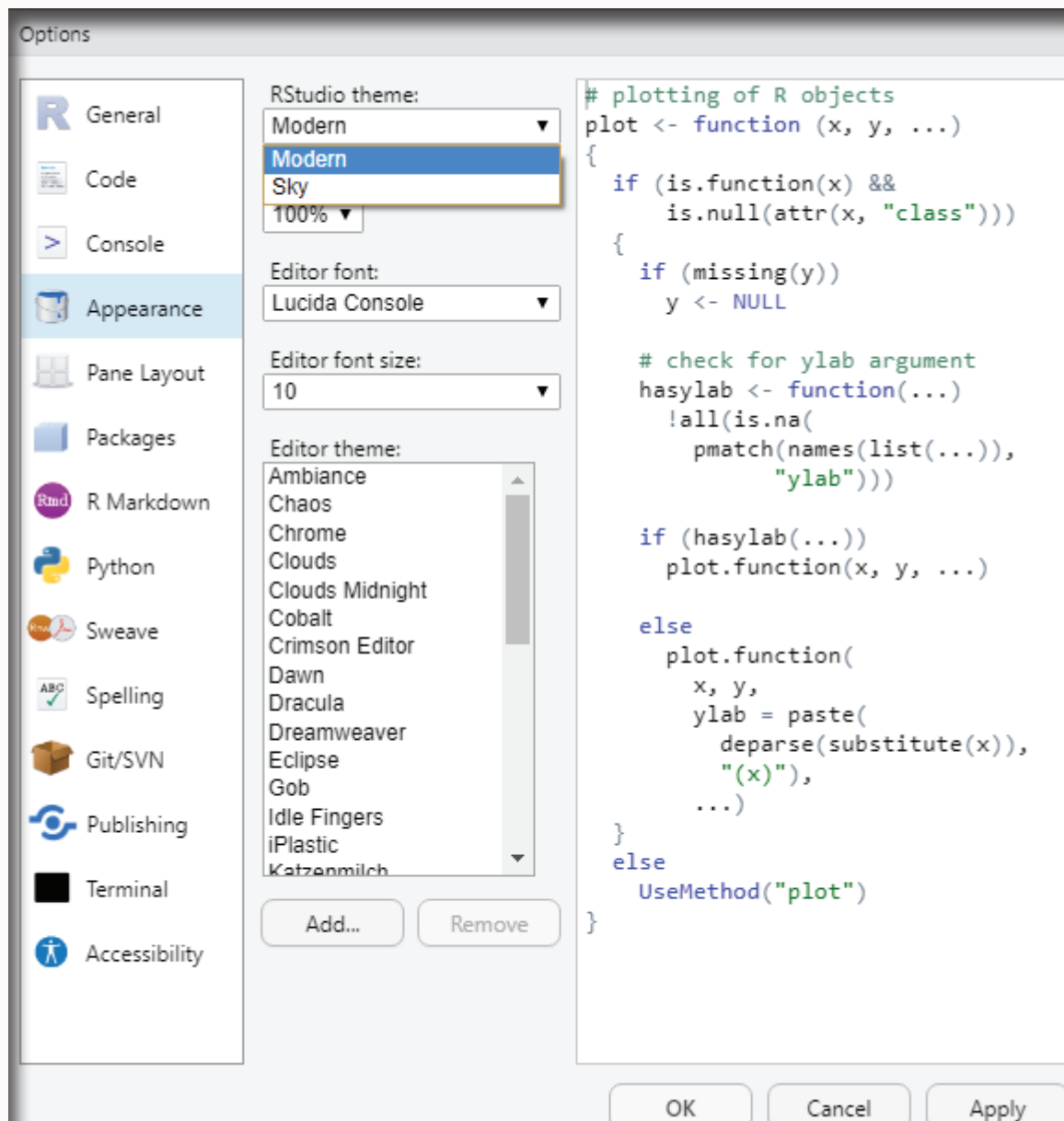


Image showing Appearance setting where RStudio themes can be changed to suit user preference.

Updating R and RStudio:

When software is being updated, one needs to update R and RStudio separately from each other. Even though R and RStudio work closely with each other, they still constitute separate pieces of software. RStudio and R cannot update on their own because some packages may not work after switching to the new version. If something goes wrong the user can still downgrade R version in RStudio. After the new version is installed, the previously installed packages will not go to next version. Extra procedures need to be performed. Upgrading R on windows could be tricky. Easiest option would be to uninstall R and then install the new version. One needs to reinstall all required packages with the new version of R and then delete the old library once they are not needed.

Updating R using installr package:

The {installr} package offers a set of R functions for the installation and updation of software. This package is available for windows OS only. The following code should be used:

installing/loading the package:

```
if(!require(installr)) {  
install.packages("installr"); require(installr)} #load / install+load installr
```

using the package:

updateR() # this will start the updating process of your R installation. It will check for newer versions, and if one is available, will guide you through the decisions you'd need to make.

Running this fuction will perform the following steps:

1. Check what is the latest R version. If the current installed R version is up-do-date, the function ends (and returns FALSE).
2. If a newer version of R is available, the user would be asked if to review the News of the latest R version in order to decide if to install the newest R or not.
3. If the user wishes to update, the function will download and install the latest R version. The next button needs to be pressed by the user.
4. Once installation is done, the user should press "any key" and the function will proceed with copying all of the packages from the old R installation into the newer R installation.
5. The user can erase all of the packages in the old R installation.
6. After the packages are moved (and the old ones probably erased), the user will get the option to update all the packages in the new version of R.

If the user wishes to upgrade R, and only want the packages to be moved and not copied then the following command is used:

installing/loading the package:

```
if(!require(installr)) { install.packages("installr"); require(installr)} #load / install+load installr
```

```
updateR(F, T, T, F, T, F, T) # install, move, update.package, quit R.
```

Another way of updating R is to simply download the newest version and run it. It will overwrite the previous version. When R is being updated the biggest challenge is that the personal library of packages dont work anymore. If the user desires to copy the personal library then it can be copied to a new location and ensuring that the new version of R recognizes it. Some users feel that it is a good time to start with a clean slate and only install packages that are needed.

Updating R studio:

RStudio can be updated from within the software. Check for Update link can be found under Help menu. It will ensure that the new version is downloaded and installed over the old version.

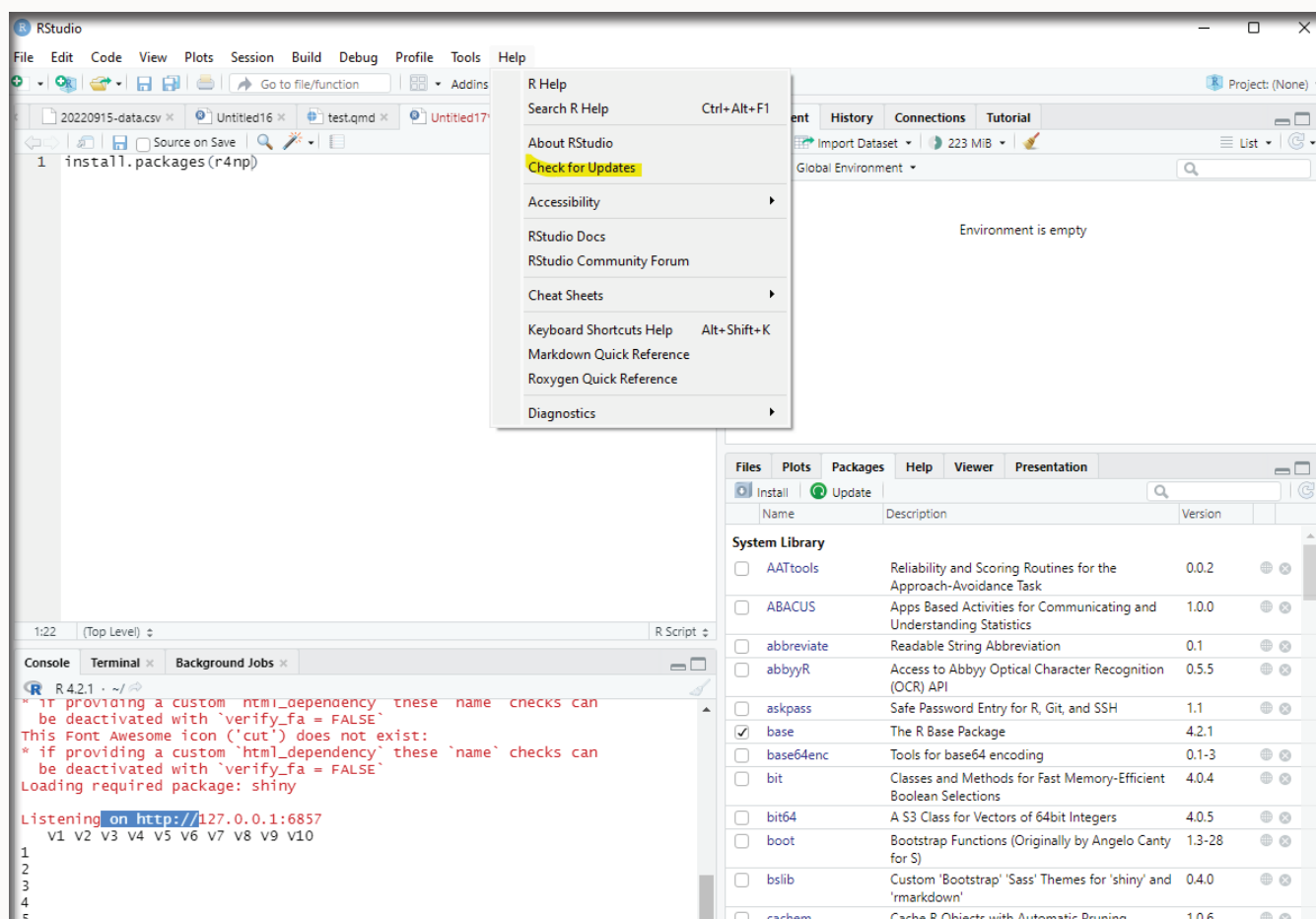


Image showing Check for Updates link under Help menu in RStudio.

Updating installed packages:

Installed packages can be updated by clicking on Check for Package updates link listed under Tools Menu. Similarly new packages can be installed by clicking on Install Package menu listed under Tools Menu. RStudio provides an easy way of updating and installing the packages desired by the user.

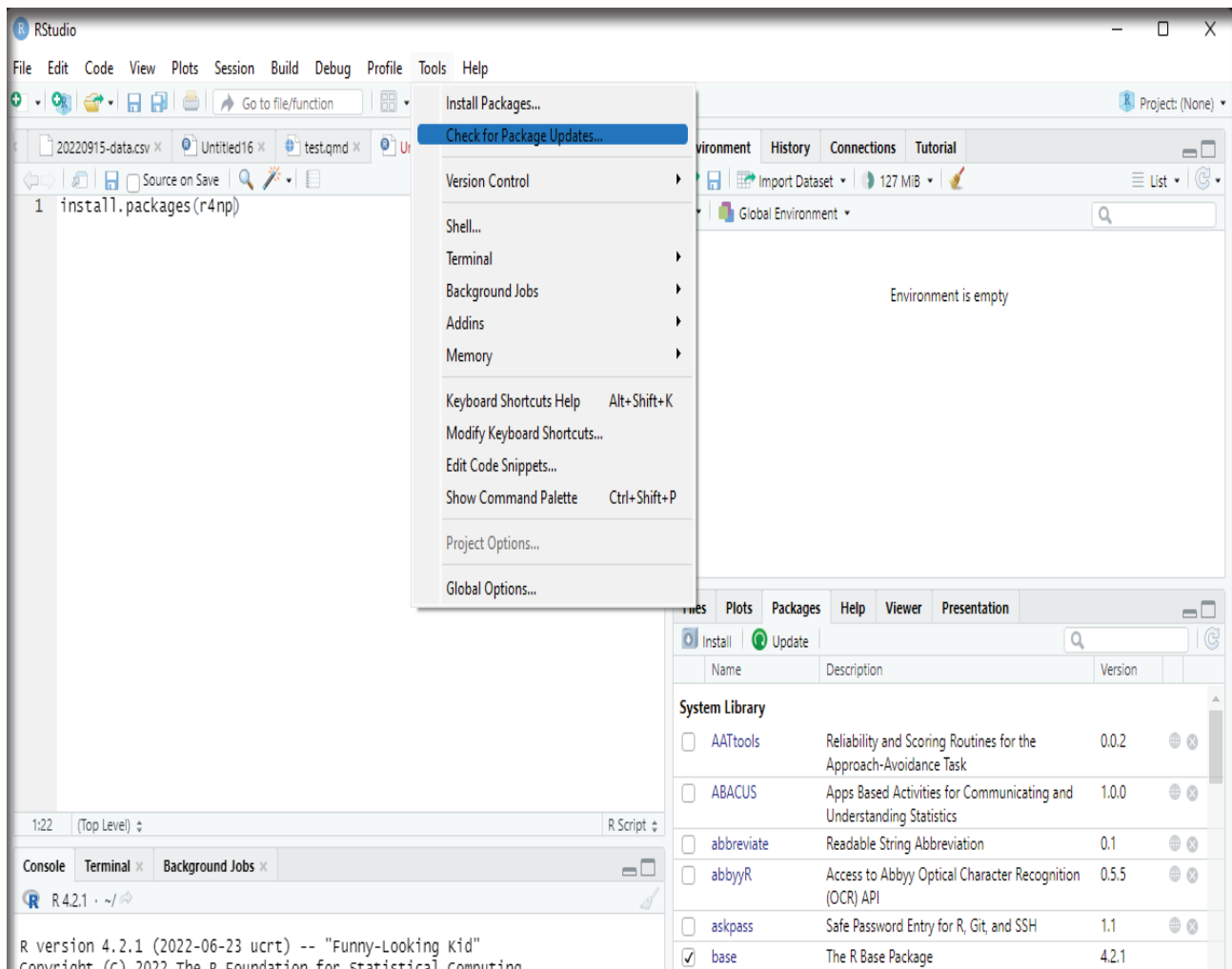


Image showing Package update link under Tools menu in RStudio that can be used to update installed packages.

RGui: (R Base software)

RGui which is the graphic user interface that is installed as part of R installation can be used to compile and run R code. It comes with a Console window where codes can be written and run. It is always better to use along with IDE like RStudio in order to make its use rather simple. Use of IDE saves a lot of time for the user. RGui can also be used for R programming without installation of IDE like RStudio. Installing RStudio along with R really makes the life of the user comfortable. User must be aware of RGui and its features. This will ensure that the user becomes a better R programmer.

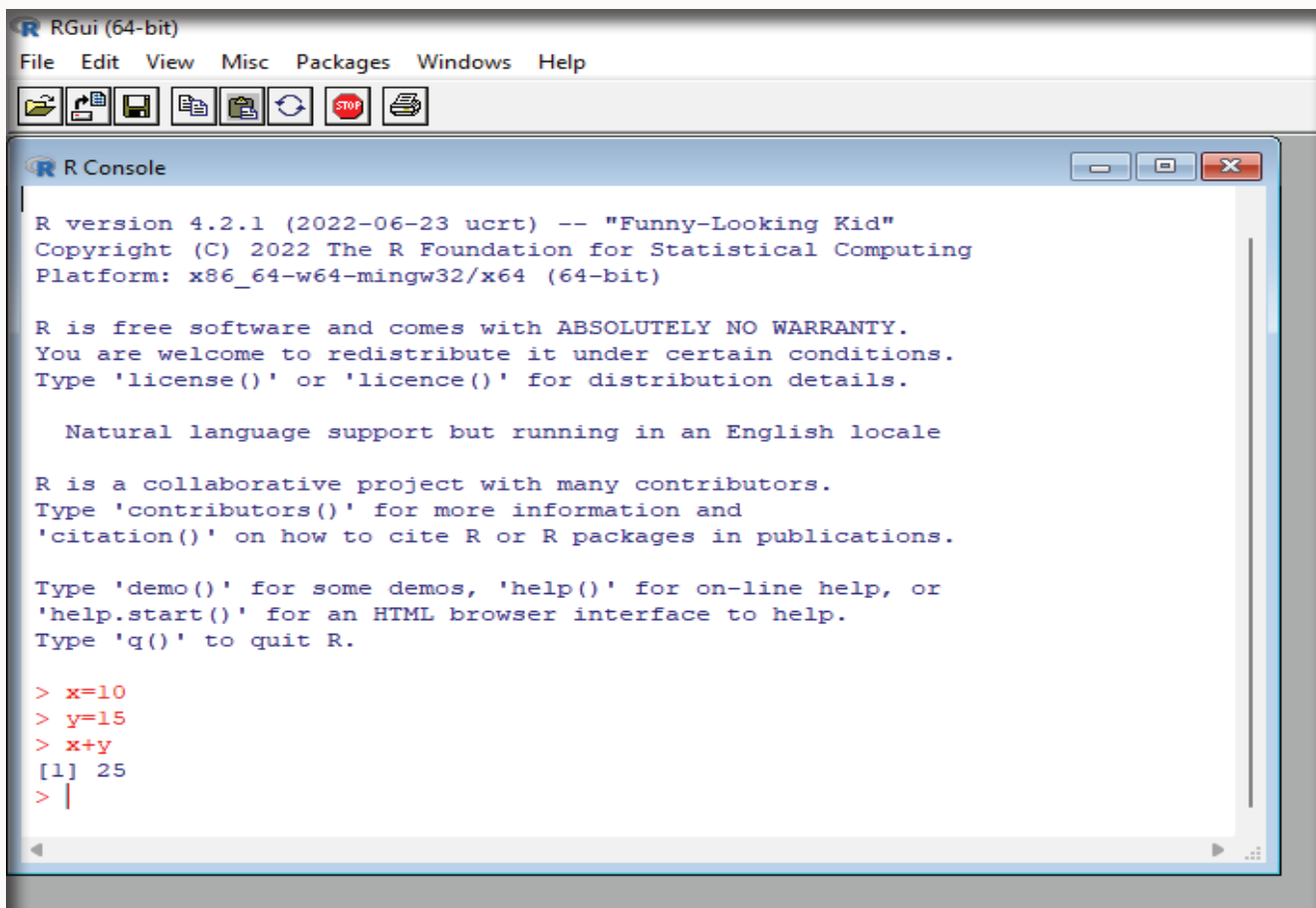


Image showing RGui interface

RGui has the following menu at the top:

File

Edit

View

Misc

Packages

Windows

Help

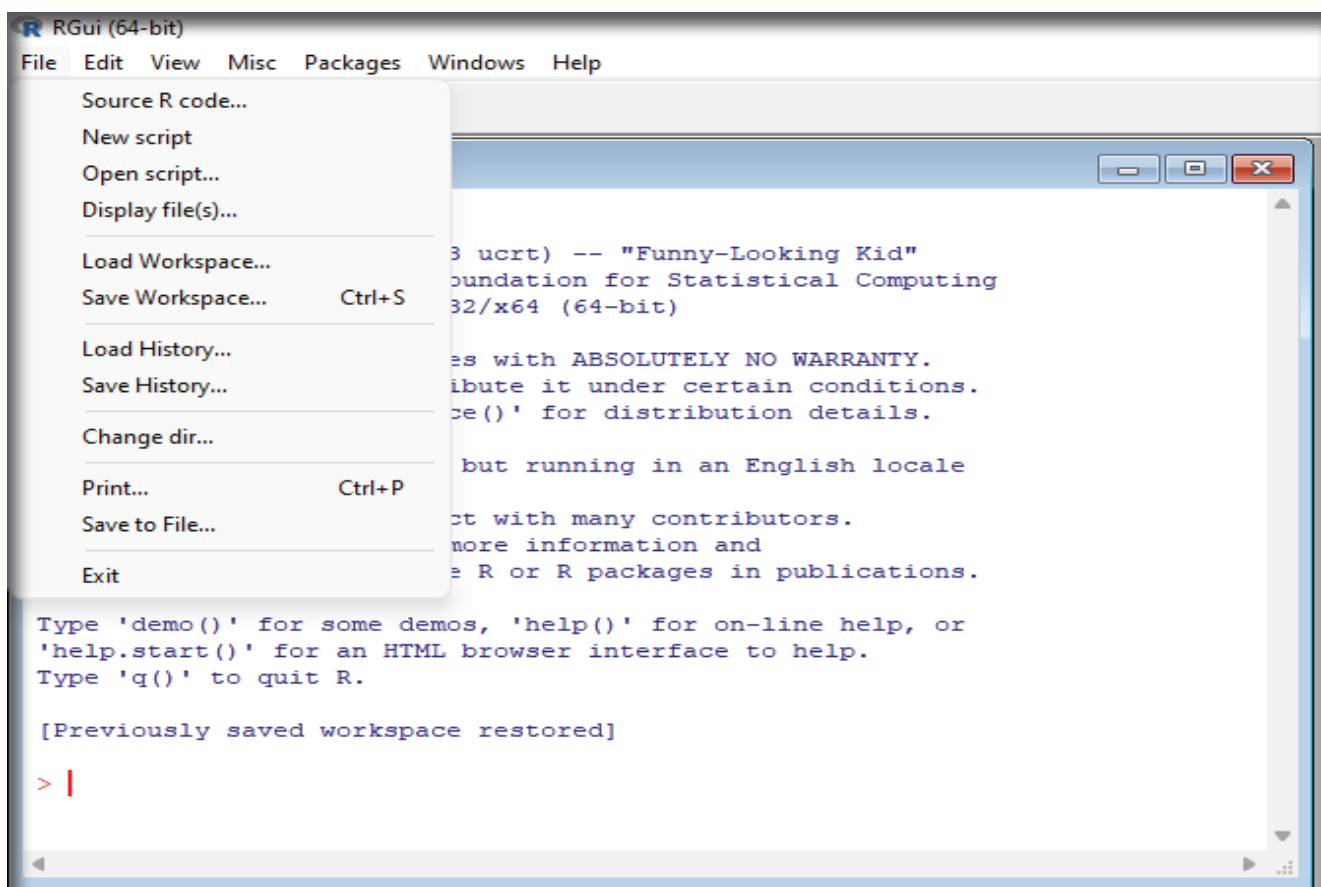


Image showing Top Menu of RGui

Under the file menu there are 12 submenus:

Source R Code - This submenu can be used to load R code file from the folder where it is stored. This can be used to reuse function that has been created in another R script. The source file causes R to accept its input from the named file. The input is read and parsed from that file until the end of the file is reached, then the parsed expressions are evaluated sequentially in the chosen environment.

New script:

To start writing a new R script in R base click on the File menu and then click on New script menu. On clicking the New script menu a R scripting window will open. Scripts can be written / typed in the scripting window and the same would be seen in the R console window.

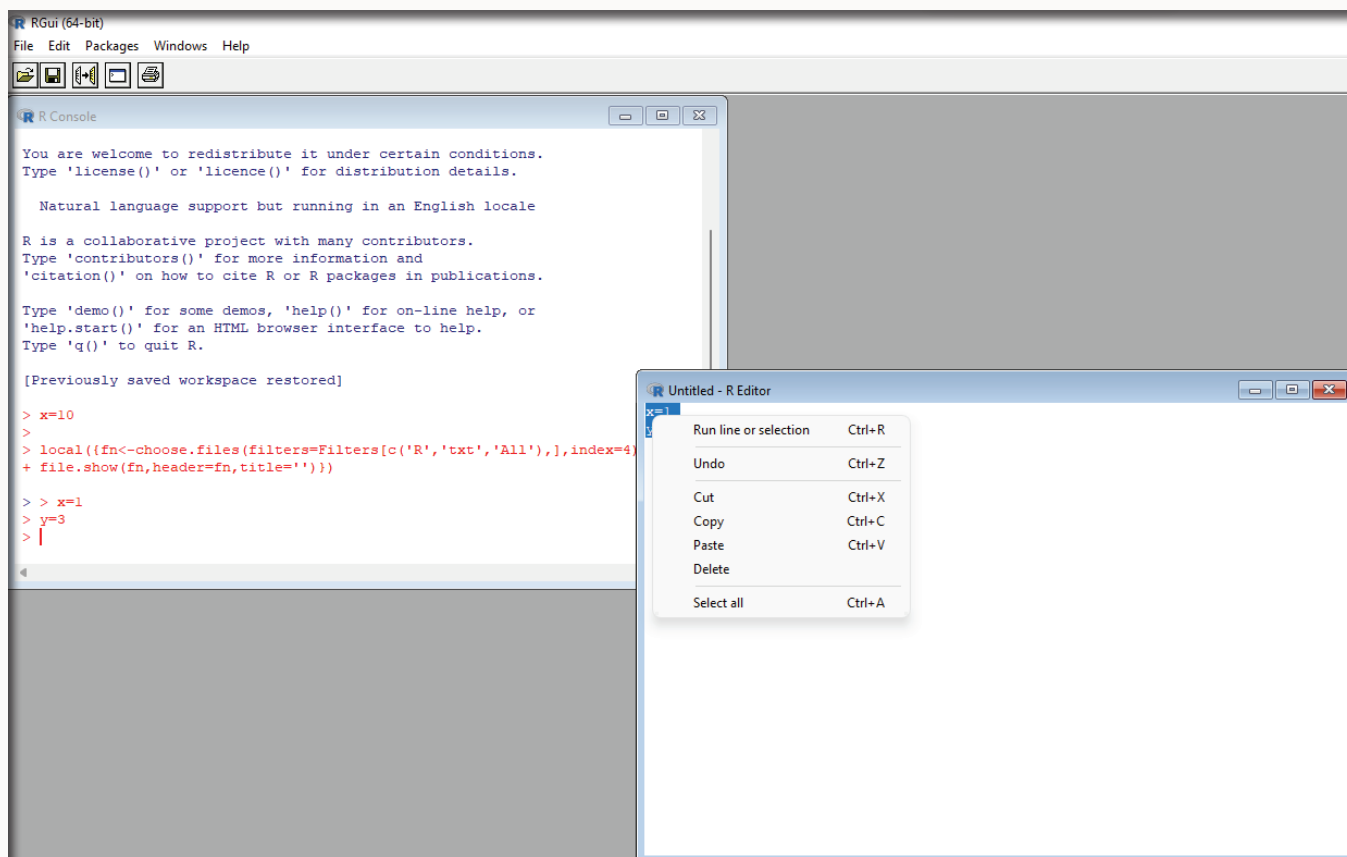


Image showing R editor opening up after the menu New script is selected and clicked.

Any script that is written in R editor will be incorporated into the console window. The code lines can be selected and on right clicking the menu as shown above will open. On choosing the code lines and clicking on Run line / selection menu the code will run in the console.

Open script menu:

This can be used to open a saved R script. Programmers usually save the script that they have created. The saved script can be opened from within R base using open script menu. On clicking Open script menu a file browser window will open from where the user can select the script that needs to be run.

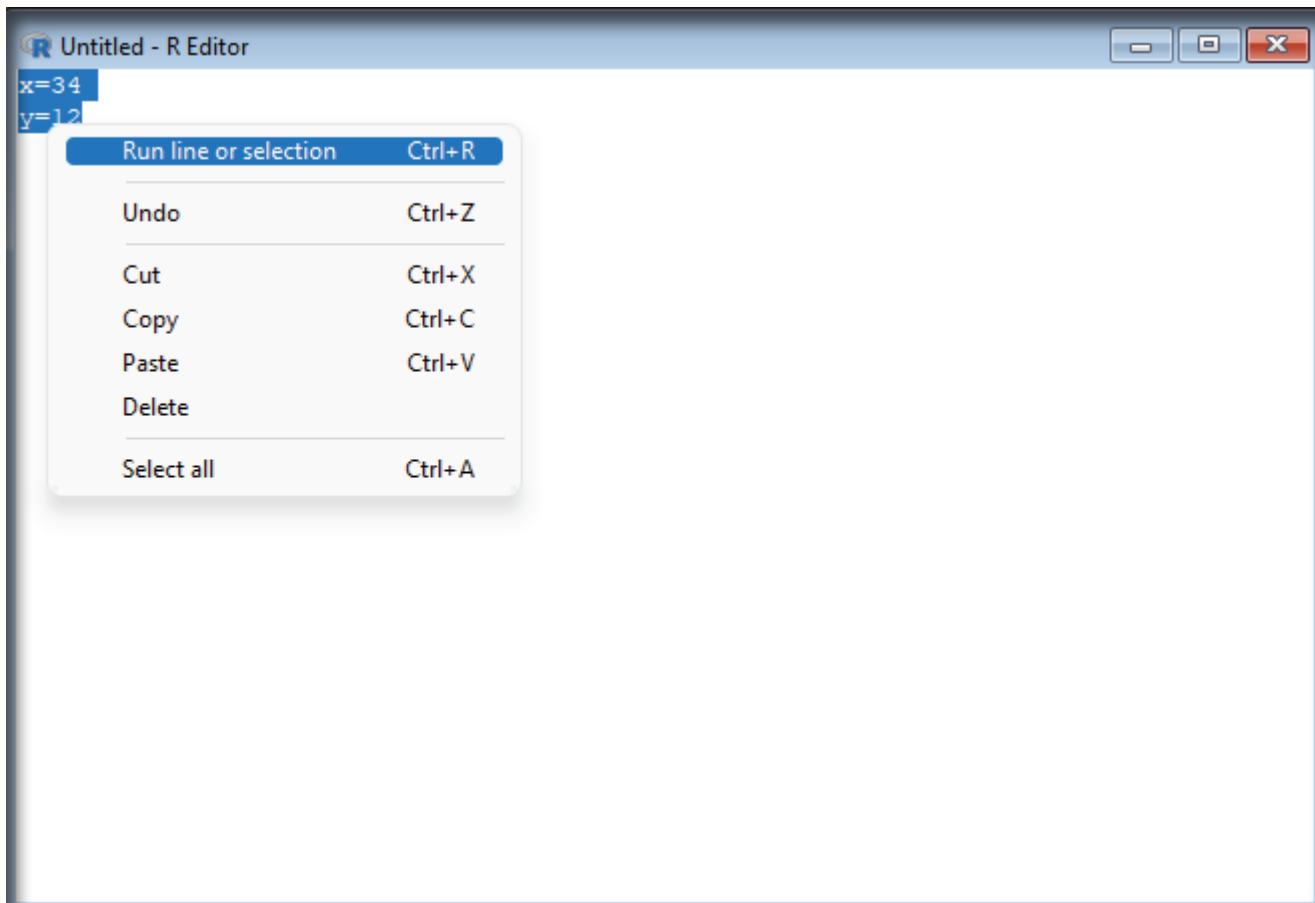


Image showing the code line that needs to be run selected and on right clicking a submenu opens up. On choosing Run line or selection the selected code runs. If undo is selected the typed code can be undone. Similarly cut / copy / paste can be used to cut, copy or paste the code. Delete menu can be used to delete the code typed. On selecting Select all menu the entire code is selected.

Display files Menu:

On clicking this drop down menu listed under File in R Base window a file browser window will open displaying the contents of my documents folder. This menu can be used to open the file browser window. Default location where R files are stored is My documents and hence this menu opens up this folder on default.

Load workspace:

On clicking this menu file browser window opens up displaying the contents of My documents folder. This is the default location where R language scripts and objects are saved as work space. These saved files can be loaded again into the R programming console by clicking on this submenu. All the objects and functions that are created by the user can be saved in a file with a suffix .RData by using the save() function or the save.image() function in the command prompt. The assigned file name goes into the bracket.

Exact command - `>save(file="d:/filename.RData")`

`>save.image("d:/filename.RData")`

These commands will be discussed in detail in later chapters.

Save workspace:

The user is prompted to save the R script as well as the objects in the console on exiting the software. The save file has a suffix of .R. The default location where the workspace is usually saved is Documents or My Documents folder as the case may be. The user of course can change the file save location when the file browser window opens up prompting the user to save the workspace.

Load History Menu:

The user can save all R commands used in an R session as .Rhistory file by using history() function. The name of the file goes between the brackets. It is important to include .Rhistory extension when saving the file at a different path. On clicking the Load History submenu a file browser will open from where the saved history file can be chosen to load into the console. R code used to save History file is `>history("d:/filename.Rhistory")`. Save history menu that is available under File Menu can also be used to save the R commands used in the console.

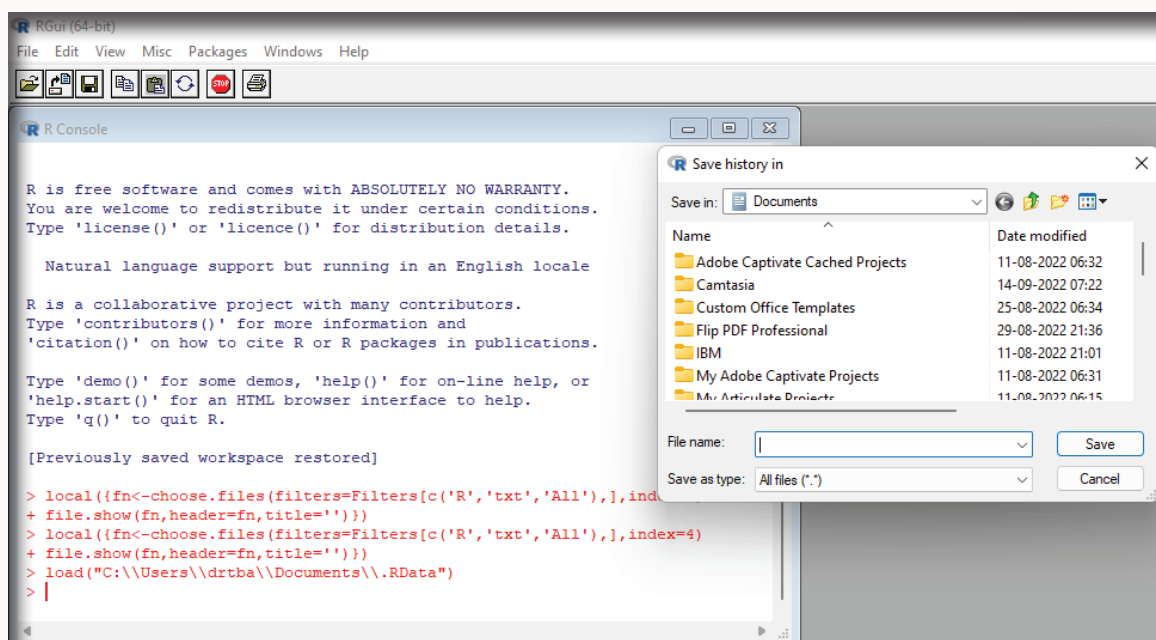


Image showing file browser window opening up on clicking Save History submenu under File menu. The user can assign a name for the file and save it. Default folder that opens is Documents.

Change dir...:

This menu on choosing opens up the file browser presenting the user with the option of changing the default working directory where the various R objects and scripts are stored.

Print:

Using the Print submenu from the File menu the user can print out the contents of the Console. If desired the contents of the console can also be printed out as a PDF.

Save to File:

This submenu can be used to save the entire session as a file. This will ensure that the user has the option of continuing from the previous session on opening the software the next day.

Exit:

On clicking this submenu, the software can be made to exit. Before exiting the software gives the user the option of saving the session.

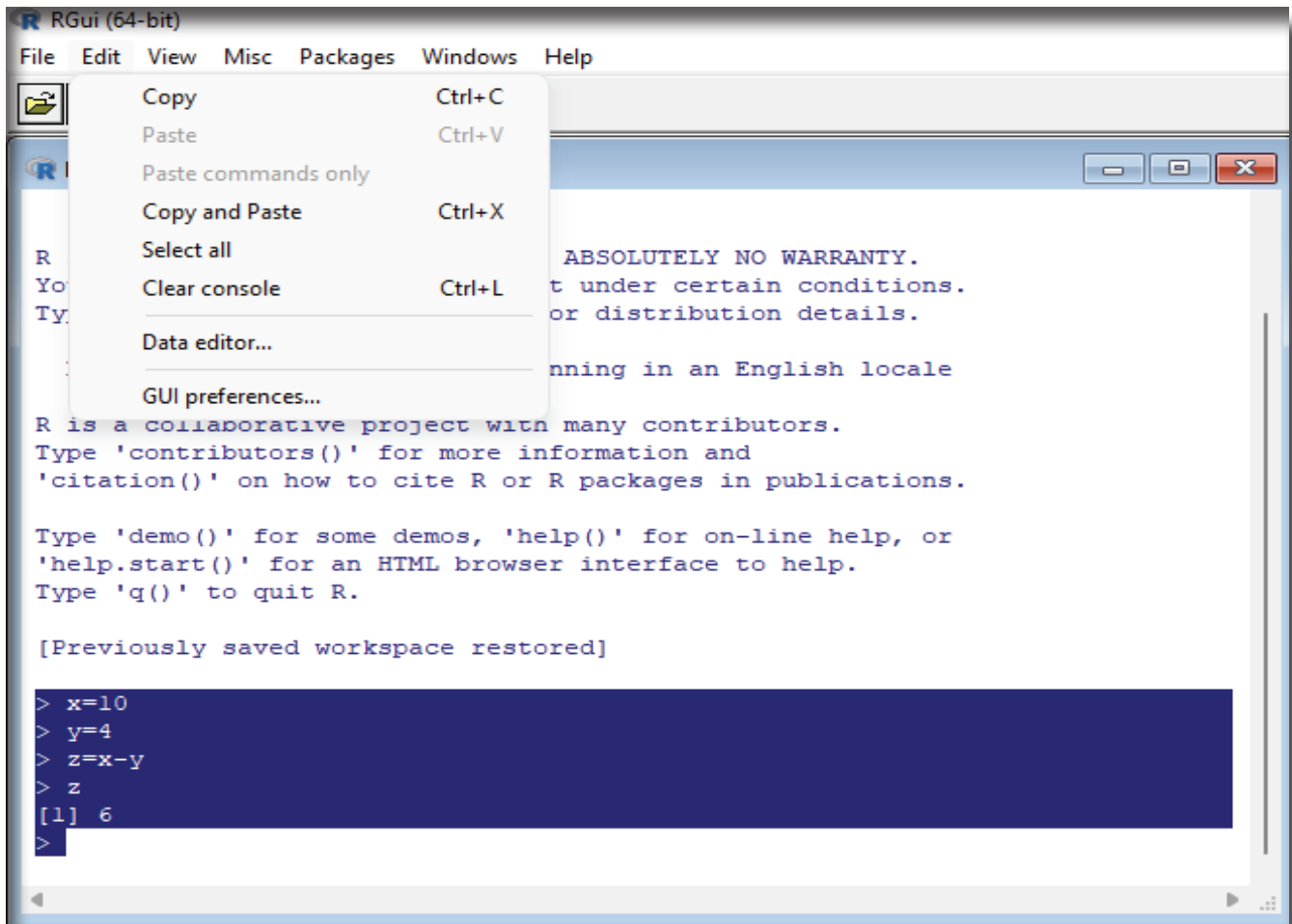


Image showing Edit menu and its various submenus

In the Edit menu the following submenu can be seen:

- Copy
- Paste
- Paste commands only
- Copy & paste
- Select all
- Clear console
- Data Editor
- Gui preferences

Copy / Paste menu can be used to copy console contents and paste them. One can choose paste commands only to paste only the commands into the console. Copy and paste menu can be chosen to do both job in one go.

Select all submenu ensures all the contents of the R console selected.

Clear console submenu clears the contents of the console.

Data Editor:

This submenu is used to edit data frame or matrix. On clicking the Data Editor submenu a window will open asking the user for the file name of the data frame / matrix that needs to be edited.

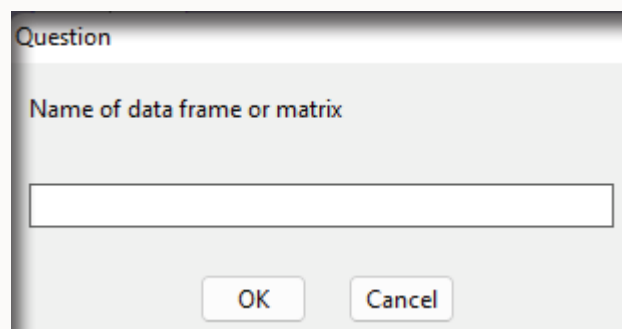


Image showing the dialog box that prompts the user to key in the name of the data frame or matrix that needs to be edited.

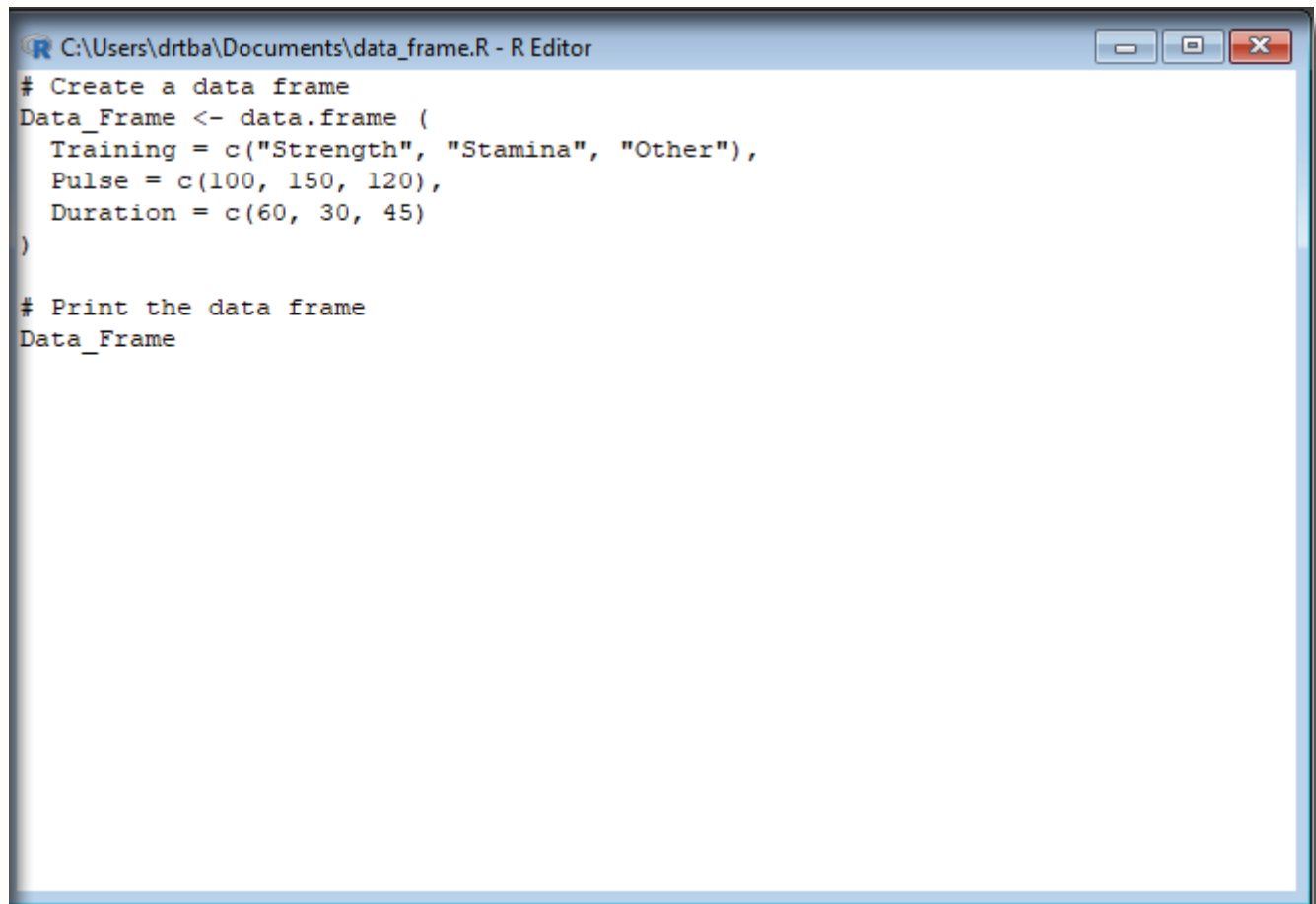


Image showing the Data editor opening after the file name of the data frame is keyed in. Using this interface data can be edited.

GUI Preferences:

This submenu opens up GUI preferences window where T console GUI settings can be manipulated. Default settings of RGUI are ideal for a normal user.

Default settings include:

Single or multiple - MDI MDI toolbar

Pager style - Multiple windows

Font - courier New True type. Size 10 with normal style.

Console rows - 20 columns - 71

Console and Pager colors can also be changed from the default white.

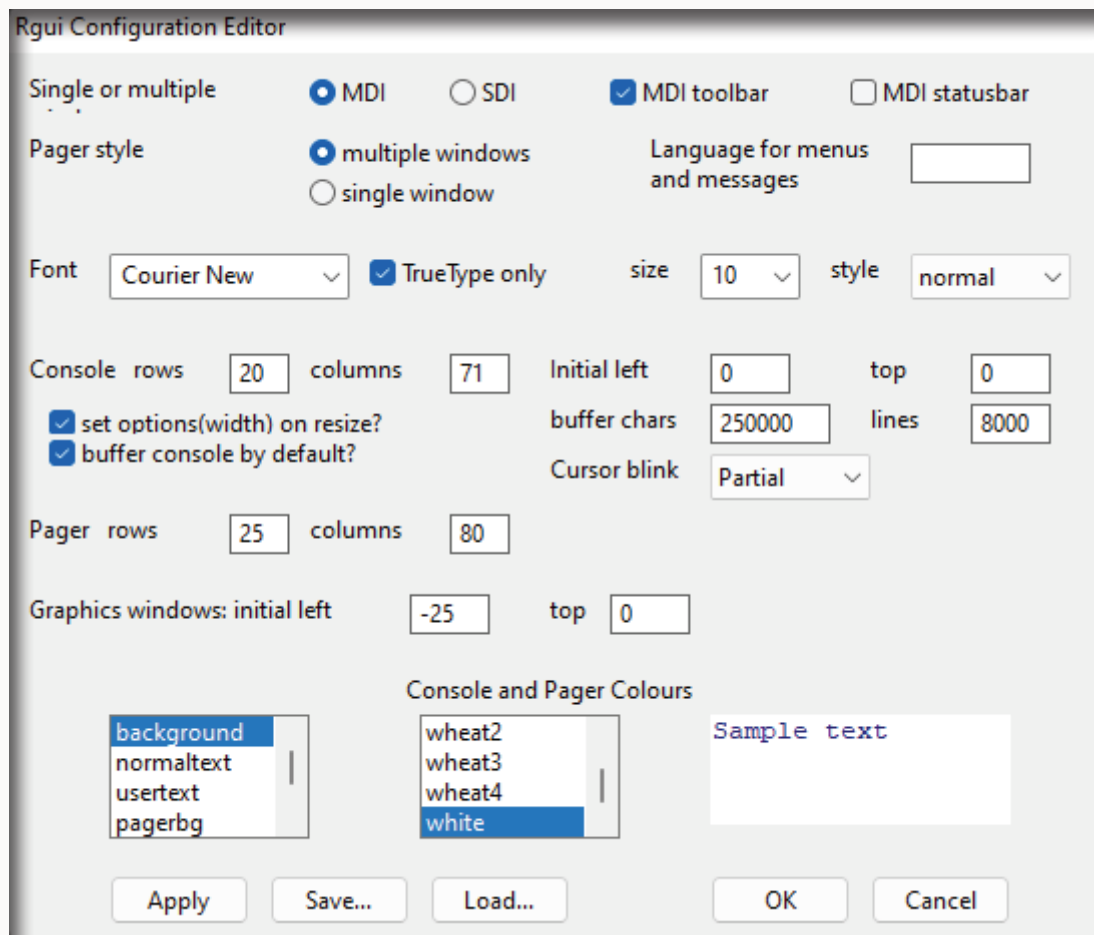


Image showing GUI preferences window.

Single or multiple - MDI is chosen since in this setting R console is displayed with menu at the top. If SDI is chosen only the R console would be opening. In this setting the top menu is not displayed. This setting can be chosen if the user desires an uncluttered environment. For the menu bar to be displayed the MDI toolbar box should be checked. If the user desires the menu to be displayed as a sidebar the MDI sidebar button should be checked instead of MDI toolbar button.

Users commonly change the font type and size to suite their preference. The next setting that is changed is the Console and Pager colors. Console and Pager colors when selected will be displayed in a small preview box. User can visualize the effect of the color settings in the preview box and decide which setting would be appropriate.

View menu:

This menu can be used to control whether the Tool bar and status bar is visible or not. If the user decides to have the Tool bar visible always then in the view menu the Tool bar should be checked. If the status bar is to be viewed then the status bar should also be checked.

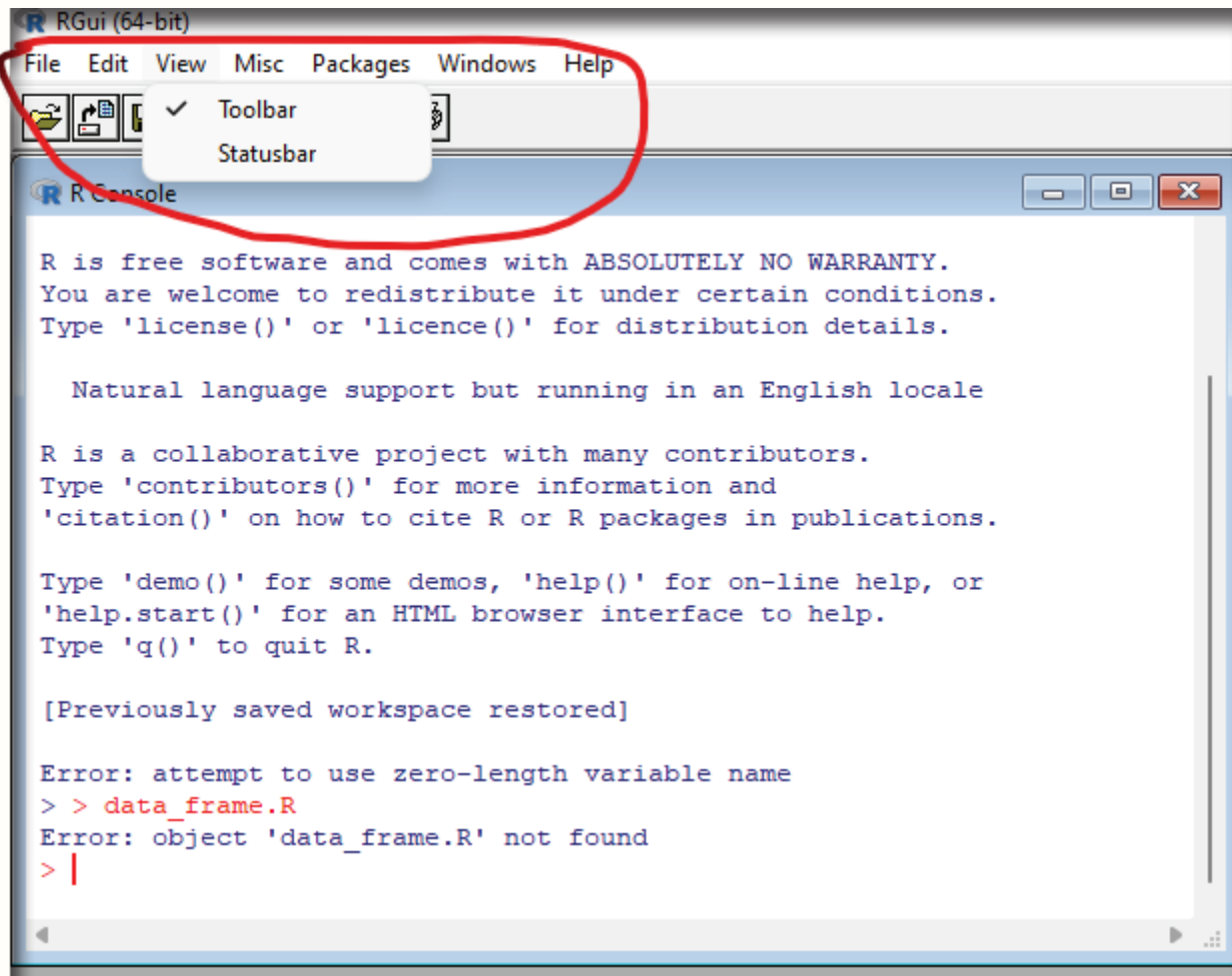


Image showing the Toolbar under view menu is selected so that the menu tools are visible.

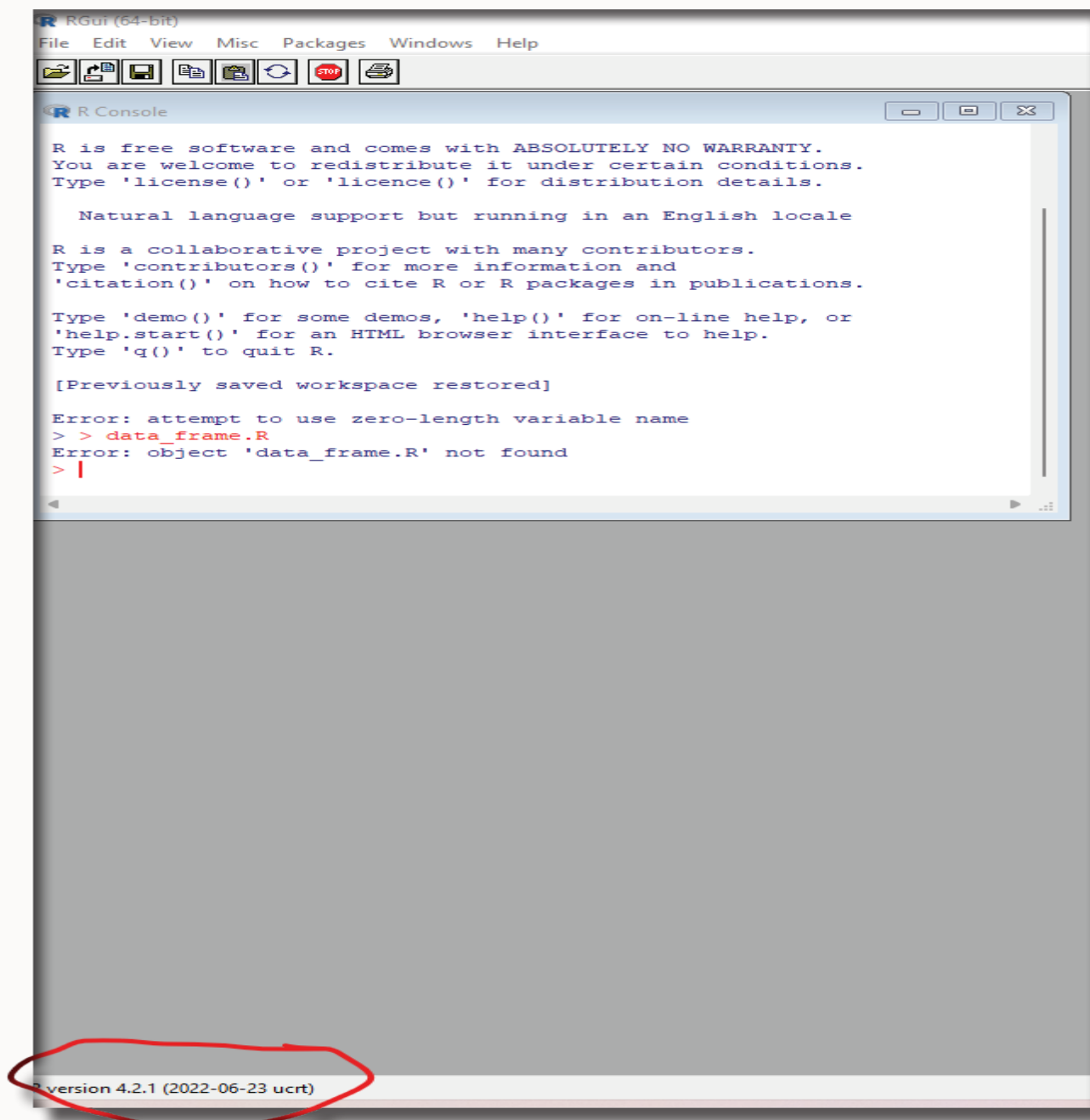


Image showing status bar visible which indicates the version of R

Misc:

Under this menu following submenus can be seen listed.

Stop current computation - Clicking on this menu will stop running R code. It would interrupt the code running process in R. One can perform the same task by pressing on Esc button of keyboard in windows machine.

Stop all computations - This submenu can be used to interrupt all running process in R.

Buffered output - An output buffer is a location in memory or cache where data ready to be seen is held until the display is ready. User can enable this function from Misc menu to ensure that the generated data by R console is displayed properly. By default this setting is enabled as shown by the tick mark before this submenu. One can choose to disable this action by clicking on the Buffered output submenu which will remove the tick mark. If the same menu is clicked again the setting will get enabled and the tick mark once again appears before this submenu. If this setting is disabled then the result will be displayed almost instantly in the console.

Word completion - This submenu is also listed under Misc menu and is enabled by default. This will ensure that when the commands are keyed into the console by the user the syntax will be auto completed. This is a rather useful setting that helps the user to save considerable amount of coding time.

File name completion - This submenu is also listed under Misc. This again is a useful tool that automatically completes the file name when the user is keying it partially. This setting is also enabled by default and saves a lot of coding time.

List objects - This submenu setting on being clicked lists all the objects in the console.

Remove all objects - This submenu when selected will remove all objects from the console.

List search path - Clicking on this submenu will list pathway of various tools and methods that can be searched.

Packages menu:

This menu contains the following submenu:

Load package - This menu can be used to load installed statistical packages and tools. If the user needs to use any package / statistical tool then they must first be loaded to the programming software. Without loading it is not possible to use the features of the package. When the package loads it also loads along with it the relevant libraries and help files to make the life of user that much comfortable. It should be stated that the sheer number of packages available could be mind boggling for the user. Many of them may not be needed for them. It is always better to install and load only the packages that are needed. There could be more than one package for performing the same function. User should be careful enough to install only those packages that are useful for their work.

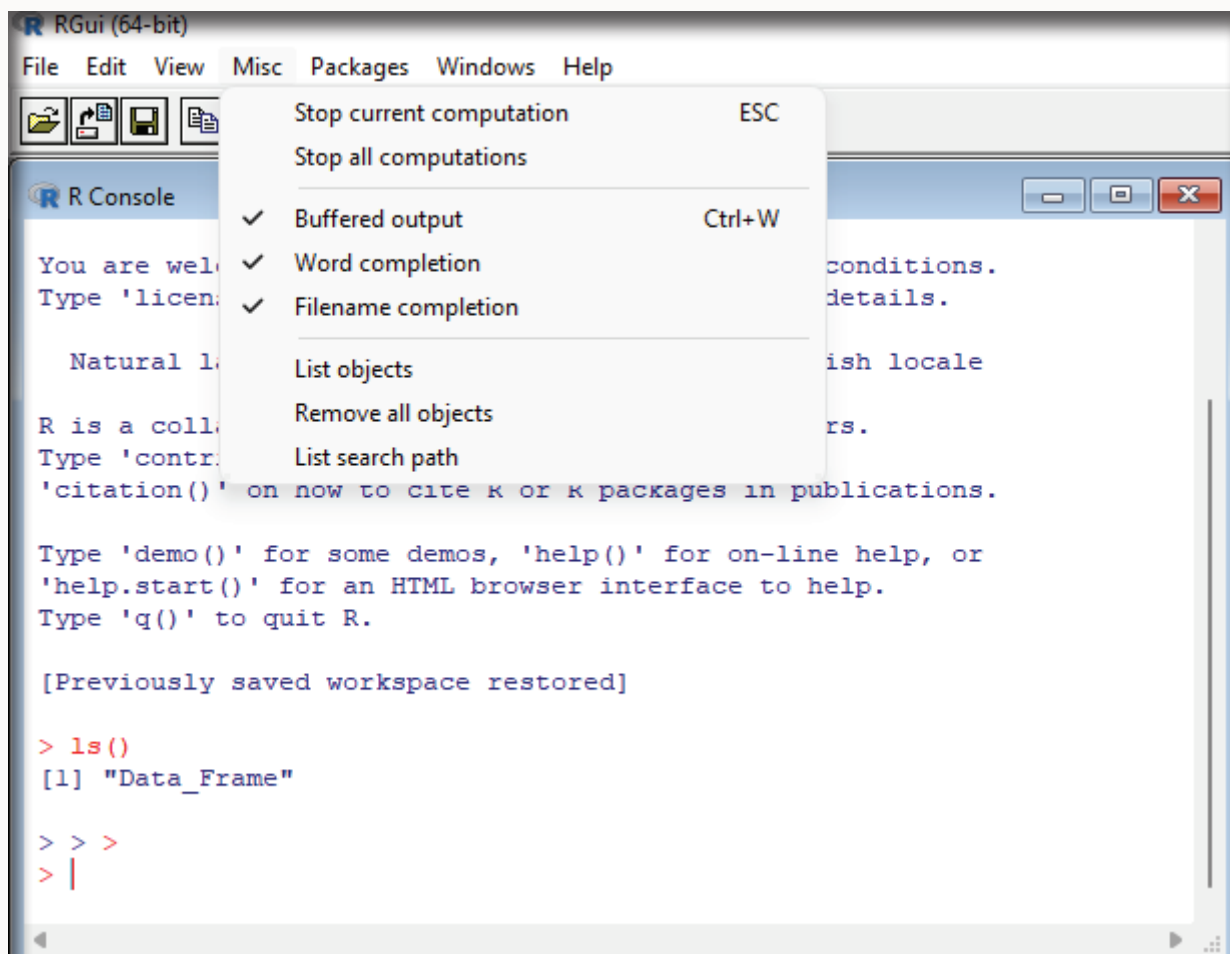


Image showing submenus listed under Misc menu

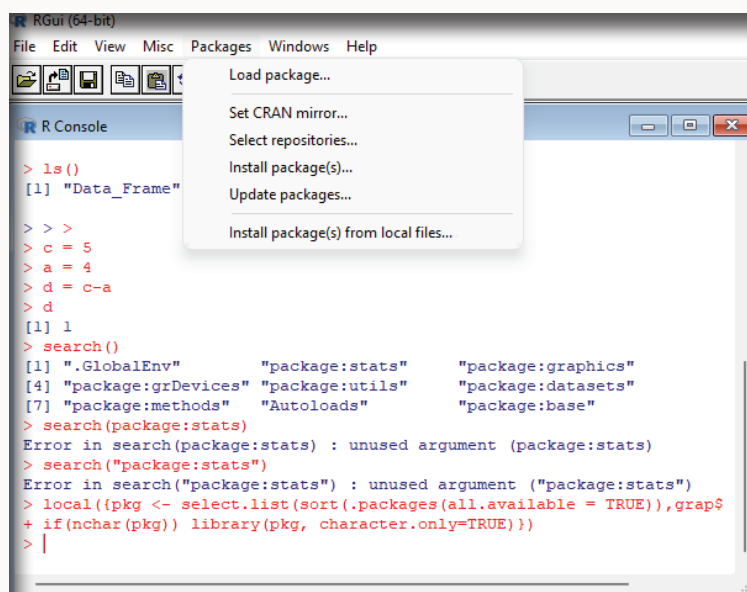


Image showing the Package menu along with its submenu

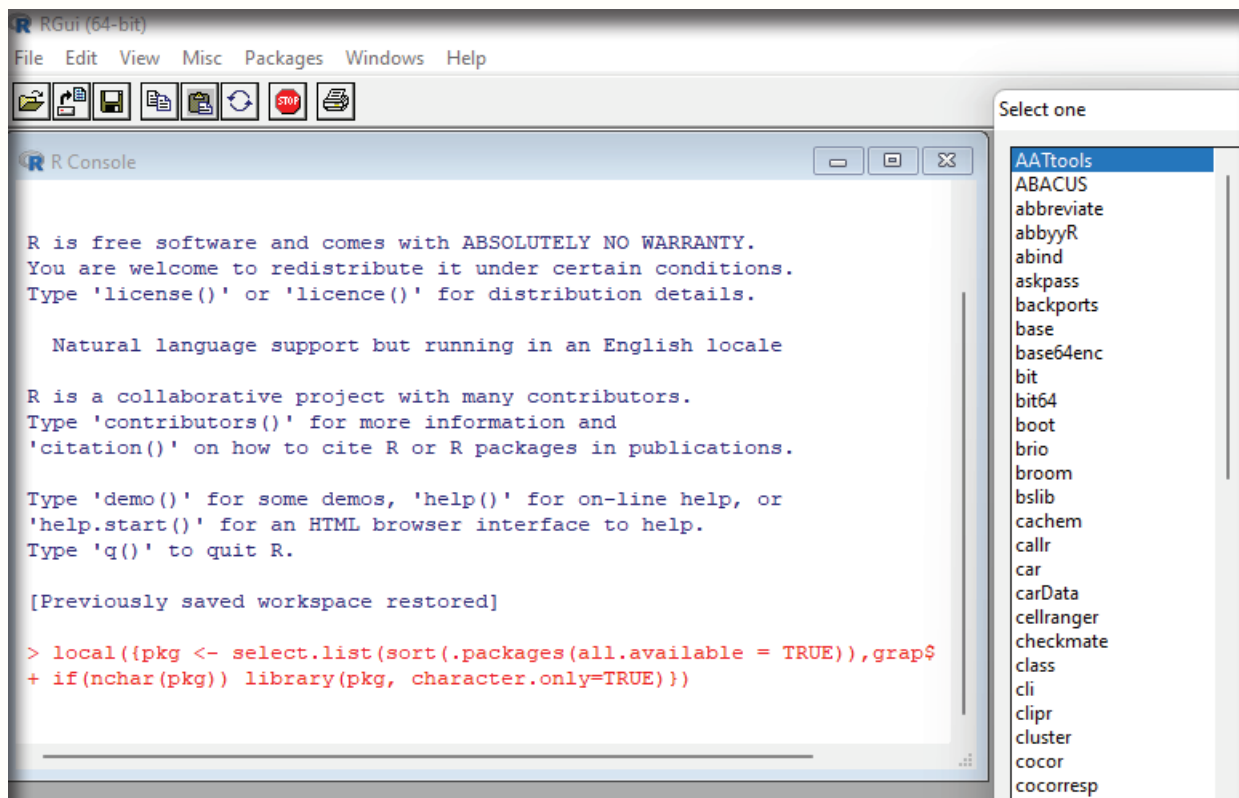


Image showing the list of installed packages that appears when the Load package submenu is clicked

Set CRAN Mirror - This submenu allows the user to set the mirror from which packages can be downloaded. The user will have to choose from the list of servers. It is ideal to choose the server that is nearest to the user so that the speed and reliability could be ideal.

Select Repositories - This submenu allows the user to select from the available Repository from which packages and other softwares can be downloaded. A repository is a central place to keep resources that the users can pull from when necessary.

Install Packages - This submenu when clicked helps the user to install packages for R. On clicking this submenu the user will be presented with a choice of secure CRAN mirrors from which download is desired. From the list the user needs to choose the optimal server. On choosing the optimal secure CRAN server the user will be presented with a list of R packages that are available for download. Download and installation will begin as soon as the user chooses the desired package and click on the OK button. Progress of the download and installation can be visualized in the console.

Update Packages - When this submenu is clicked it will display a list of available updates for the software packages installed. The user can select the packages that need to be updated and click OK for the update process to begin.

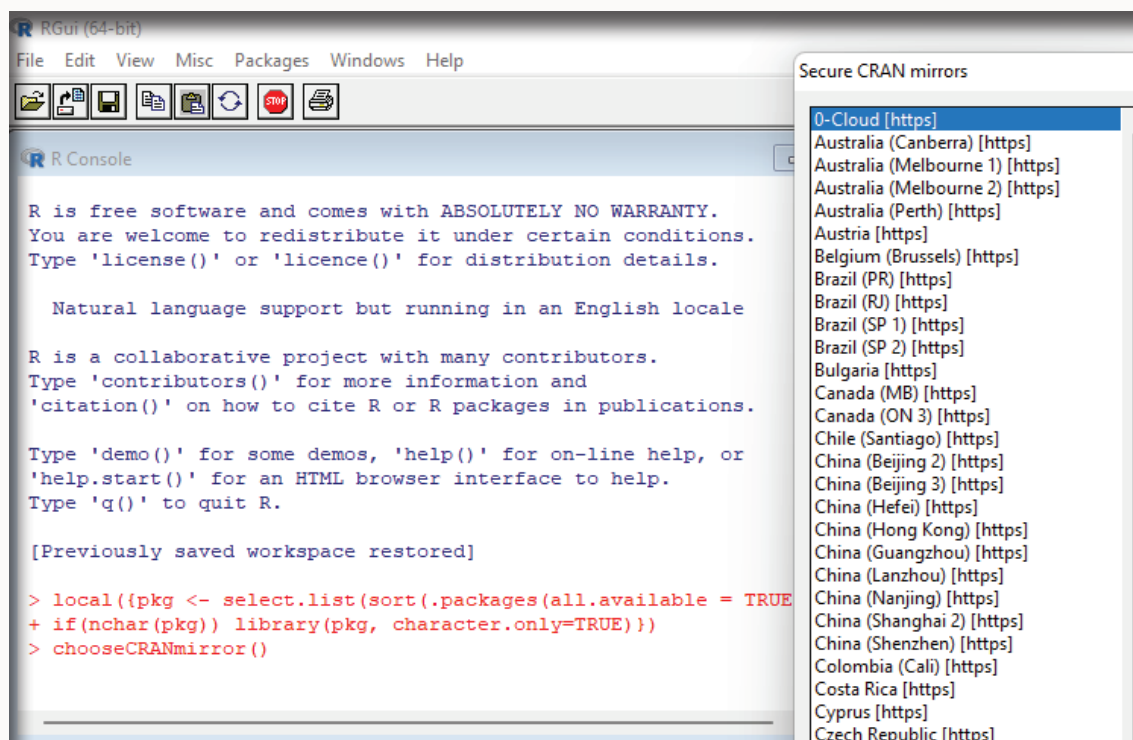


Image showing a list of CRAN mirrors (truncated) from which the ideal one can be chosen by the user. This dialog box appears when the user clicks on set CRAN Mirror submenu

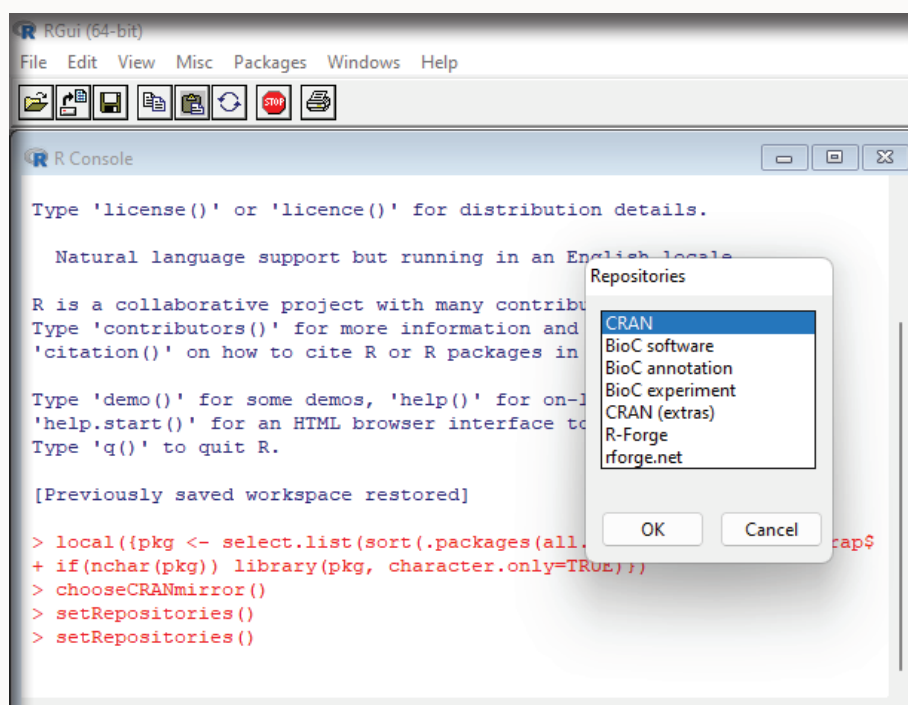


Image showing the list of Repositories from where the user can choose the desired one

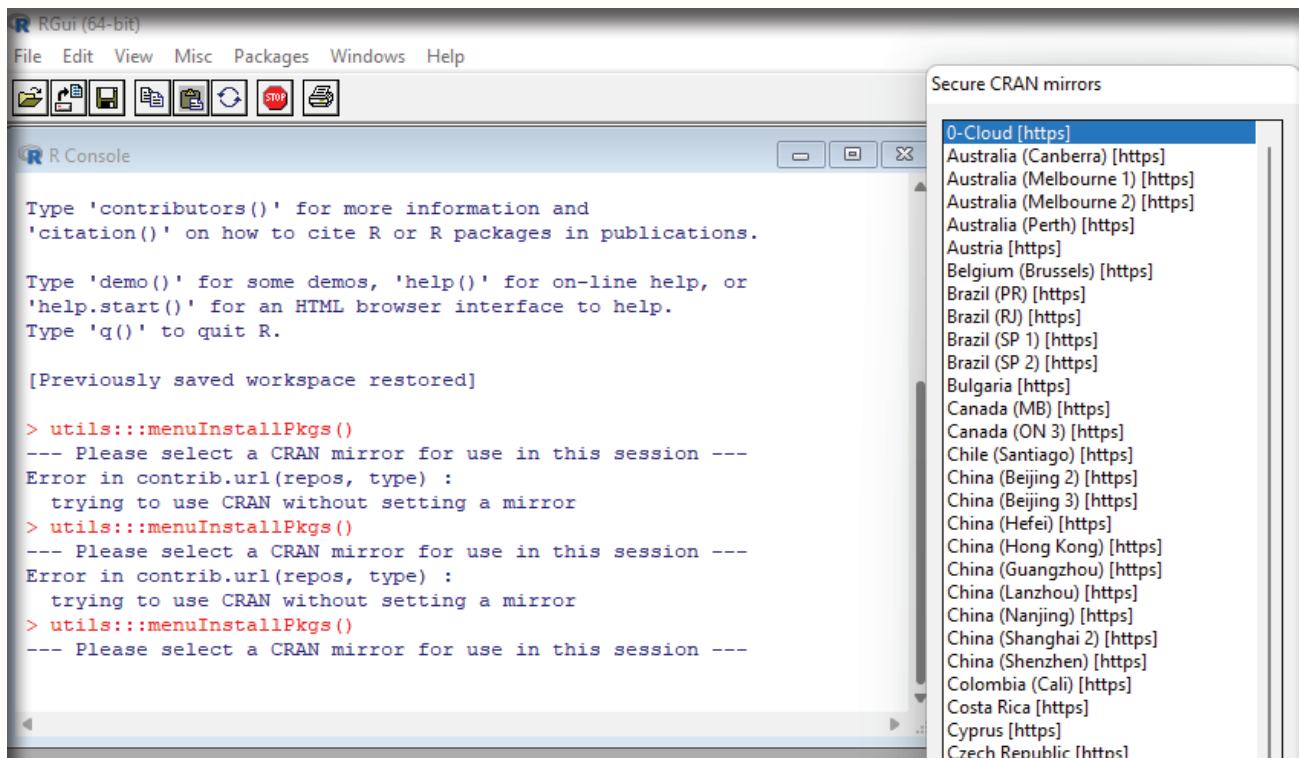


Image showing the list (truncated) list of secure CRAN mirrors from where R packages can be downloaded

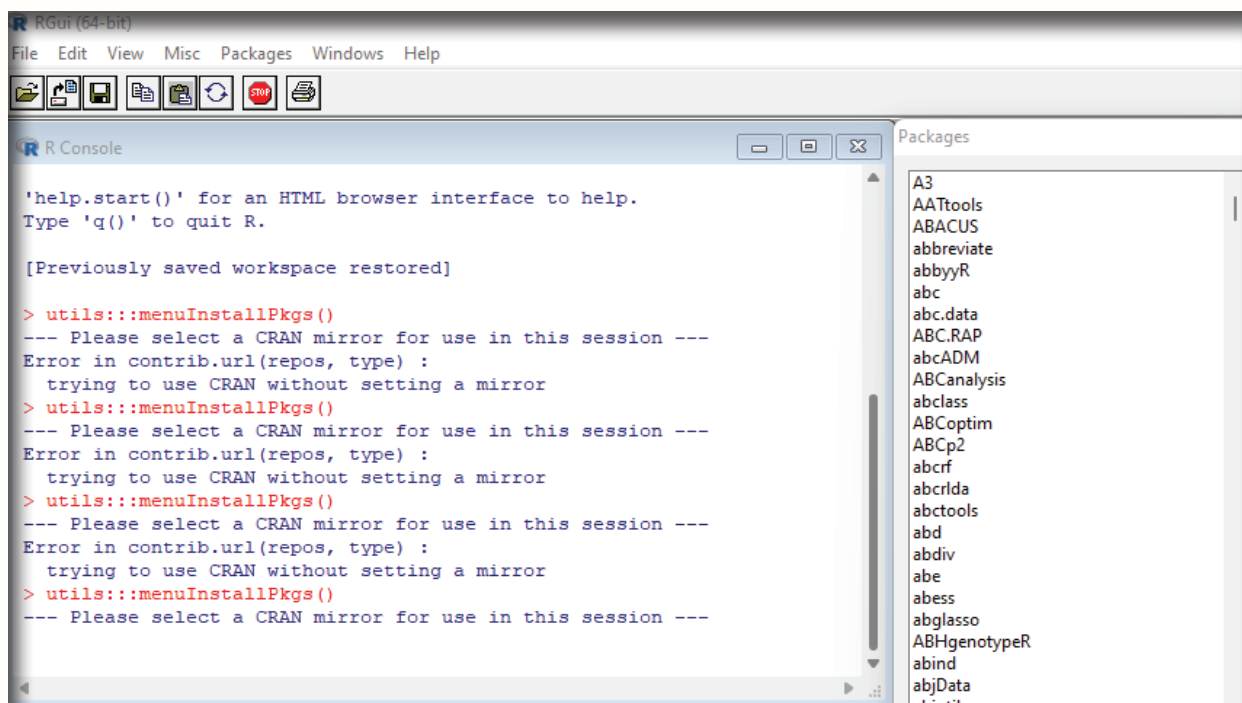


Image showing the packag list (truncated) from where the user can choose the desired one

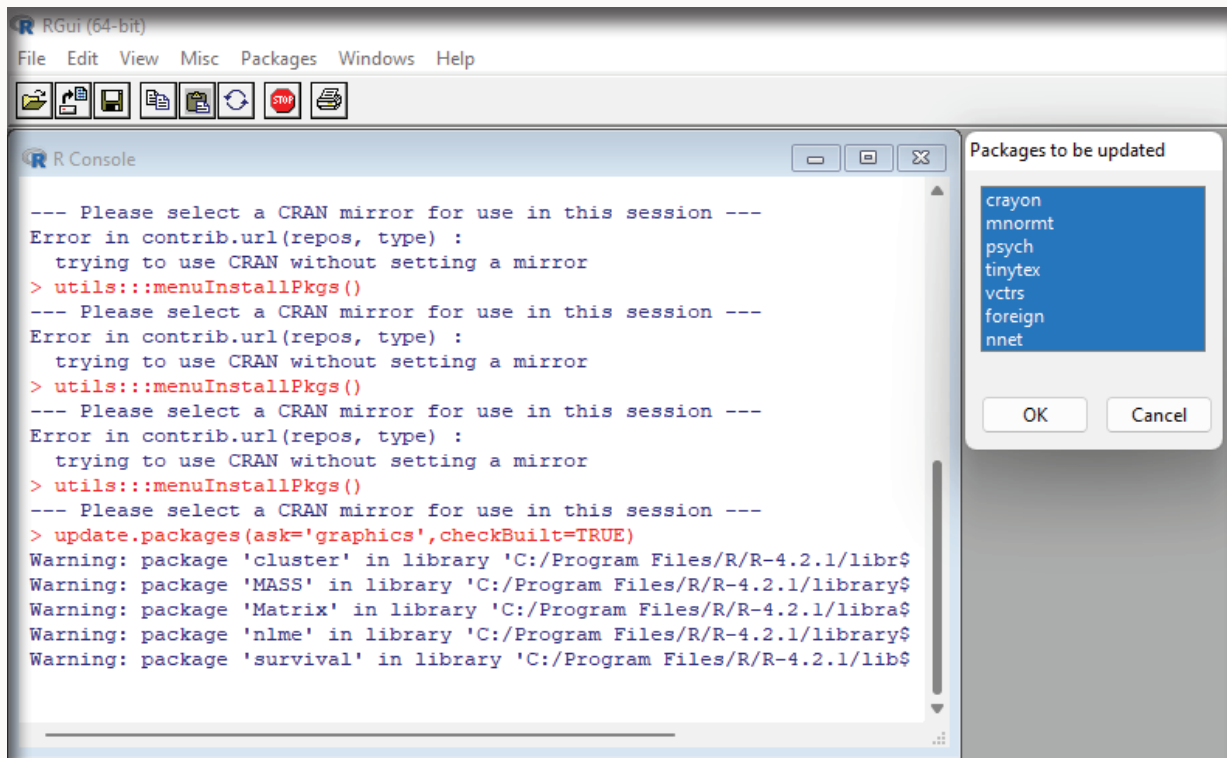


Image showing the list of packages for which updates are available

Install Package(s) from local File - This submenu when selected will facilitate the user to install downloaded package from a location in the hard disk.

Windows Menu :

This menu on clicking will reveal the following drop down submenus.

Cascade - If this is clicked the R console will assume less screen space.

Tile Horizontally - If this is clicked the R console will occupy more horizontal screen space. The console window will enlarge horizontally.

Tile Vertically - If this is clicked the R console will occupy more vertical screen space. The console window would enlarge vertically.

Arrange icons - This submenu will allow the user to rearrange icons that are present above the console window.

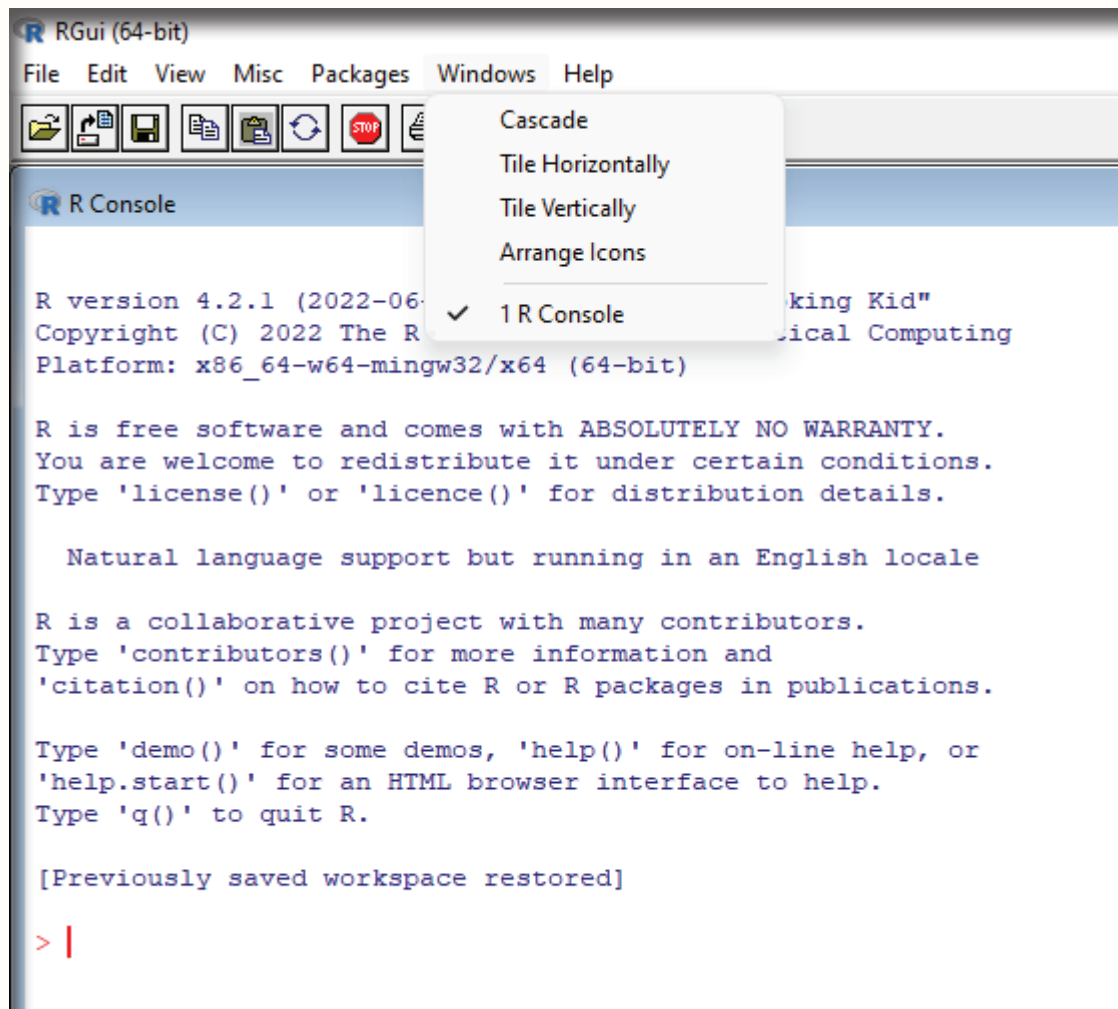


Image showing submenus under Windows menu

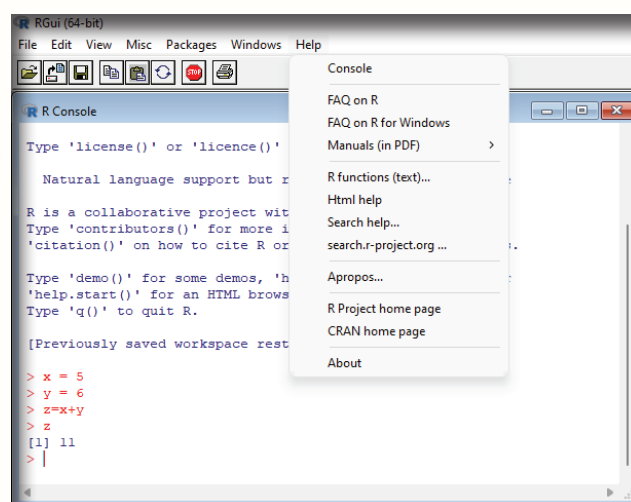


Image showing submenu listed under Help menu

Help Menu:

Under this menu various help sources and files are listed. Submenu under this main menu include:

Console - When this submenu is clicked it opens up a window containing help pertaining to Console features. It includes keyboard shortcuts for various functions of the console.

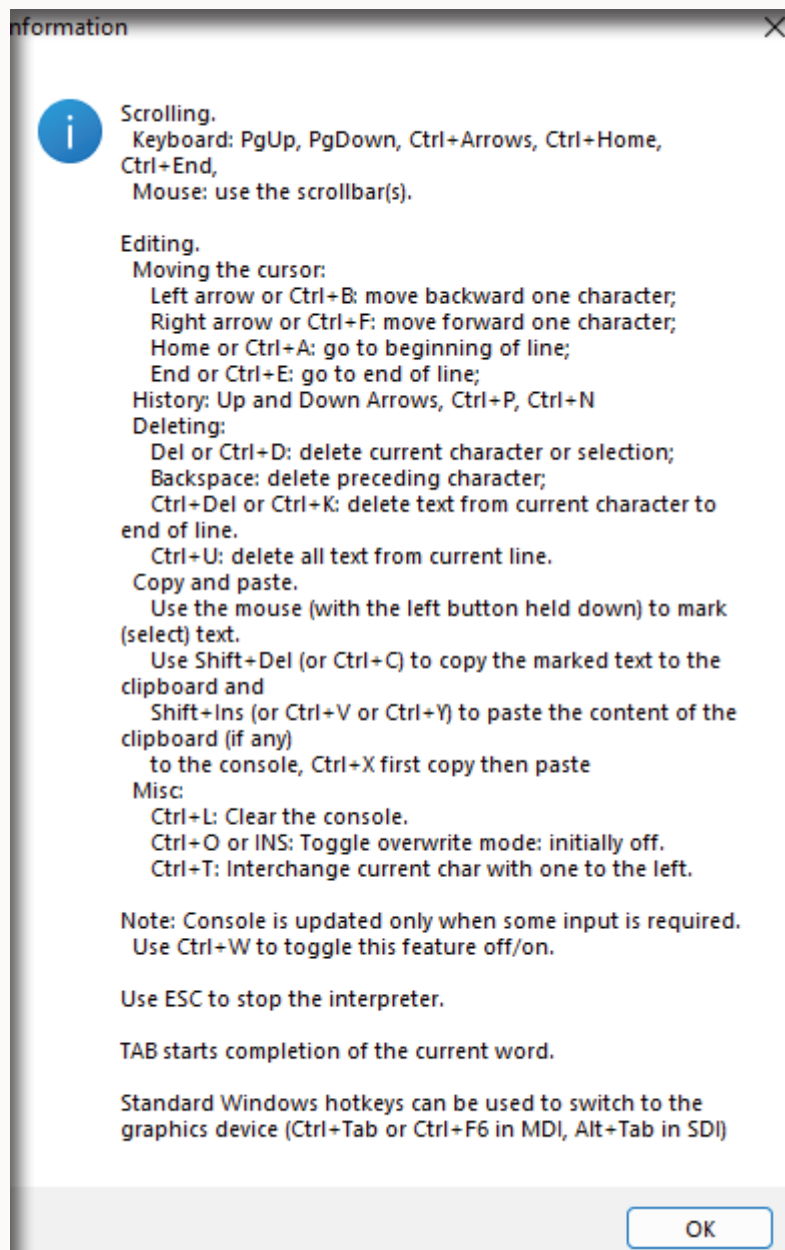
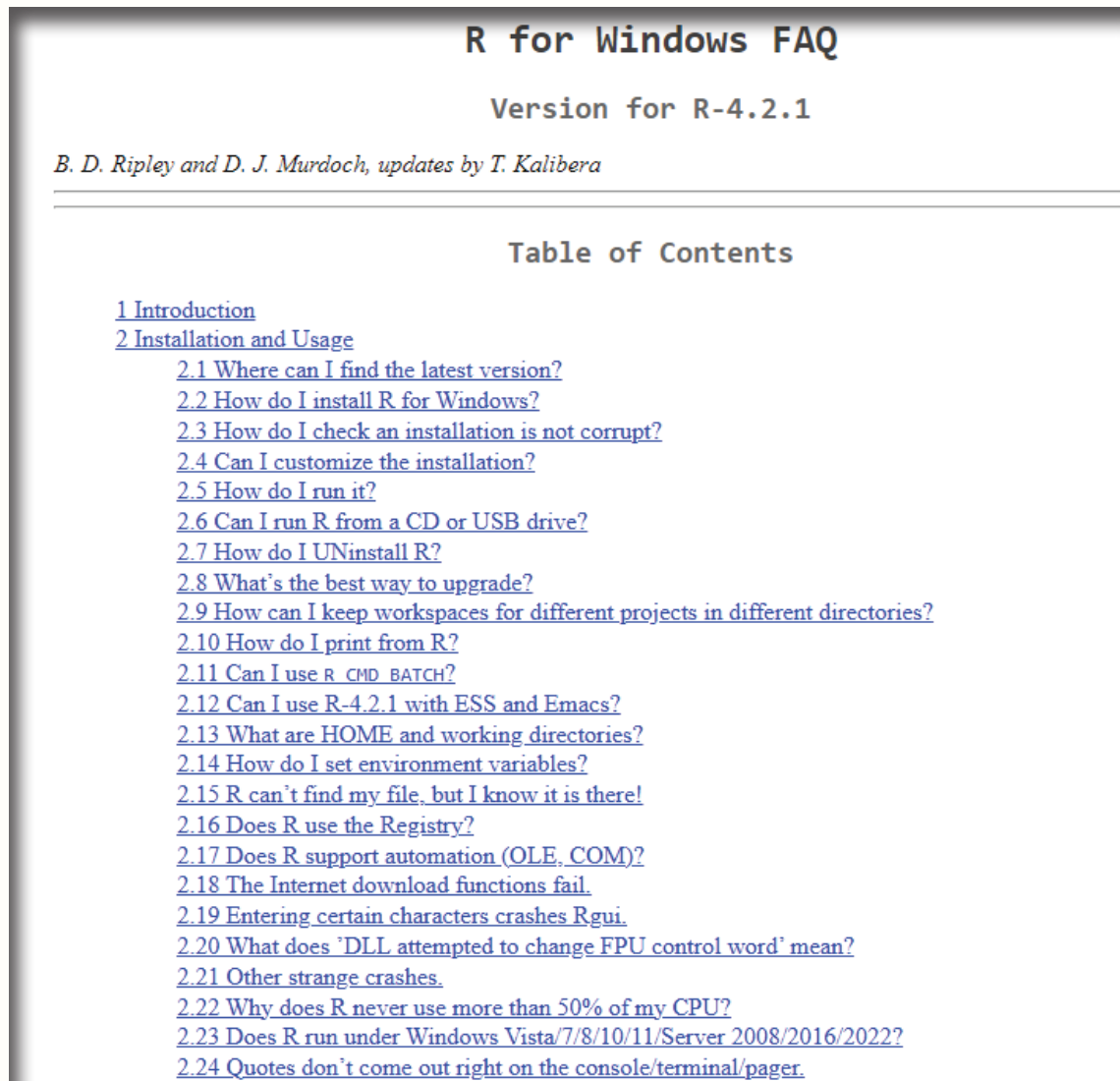


Image showing help tips pertaining to Console

FAQ on R - This submenu when clicked will take the user to a webpage displaying a set of Frequently asked questions on R and their responses.

FAQ on R for windows - This submenu on being clicked will take the user to a webpage containing various frequently asked questions pertaining to R software in windows.

The image is a screenshot of a web browser displaying the 'R for Windows FAQ' page. The page has a white background with a black border. At the top, the title 'R for Windows FAQ' is centered in a large, bold, black font. Below it, 'Version for R-4.2.1' is centered in a smaller, bold, black font. Underneath, the authors 'B. D. Ripley and D. J. Murdoch, updates by T. Kalibera' are listed in a smaller, italicized black font. A horizontal line separates the header from the main content. The main content is titled 'Table of Contents' in a bold black font. Below this, there are two main sections: '1 Introduction' and '2 Installation and Usage'. Section 2 contains 24 numbered links, each starting with '2.x' followed by a question. The links are underlined and in a blue color. The questions cover topics like finding the latest version, installation, customization, running R, uninstalling, upgrading, workspaces, printing, using R CMD BATCH, ESS and Emacs, HOME and working directories, environment variables, file finding, registry, automation, download functions, character crashes, DLL errors, CPU usage, Windows versions, and console output.

R for Windows FAQ

Version for R-4.2.1

B. D. Ripley and D. J. Murdoch, updates by T. Kalibera

Table of Contents

[1 Introduction](#)

[2 Installation and Usage](#)

- [2.1 Where can I find the latest version?](#)
- [2.2 How do I install R for Windows?](#)
- [2.3 How do I check an installation is not corrupt?](#)
- [2.4 Can I customize the installation?](#)
- [2.5 How do I run it?](#)
- [2.6 Can I run R from a CD or USB drive?](#)
- [2.7 How do I UNinstall R?](#)
- [2.8 What's the best way to upgrade?](#)
- [2.9 How can I keep workspaces for different projects in different directories?](#)
- [2.10 How do I print from R?](#)
- [2.11 Can I use R CMD BATCH?](#)
- [2.12 Can I use R-4.2.1 with ESS and Emacs?](#)
- [2.13 What are HOME and working directories?](#)
- [2.14 How do I set environment variables?](#)
- [2.15 R can't find my file, but I know it is there!](#)
- [2.16 Does R use the Registry?](#)
- [2.17 Does R support automation \(OLE, COM\)?](#)
- [2.18 The Internet download functions fail.](#)
- [2.19 Entering certain characters crashes Rgui.](#)
- [2.20 What does 'DLL attempted to change FPU control word' mean?](#)
- [2.21 Other strange crashes.](#)
- [2.22 Why does R never use more than 50% of my CPU?](#)
- [2.23 Does R run under Windows Vista/7/8/10/11/Server 2008/2016/2022?](#)
- [2.24 Quotes don't come out right on the console/terminal/pager.](#)

Image showing R for windows FAQ web page that gets displayed when this submenu is clicked

Manuals in (PDF) - On clicking this submenu user will be presented with the choice of links to various manuals for better understanding of R.

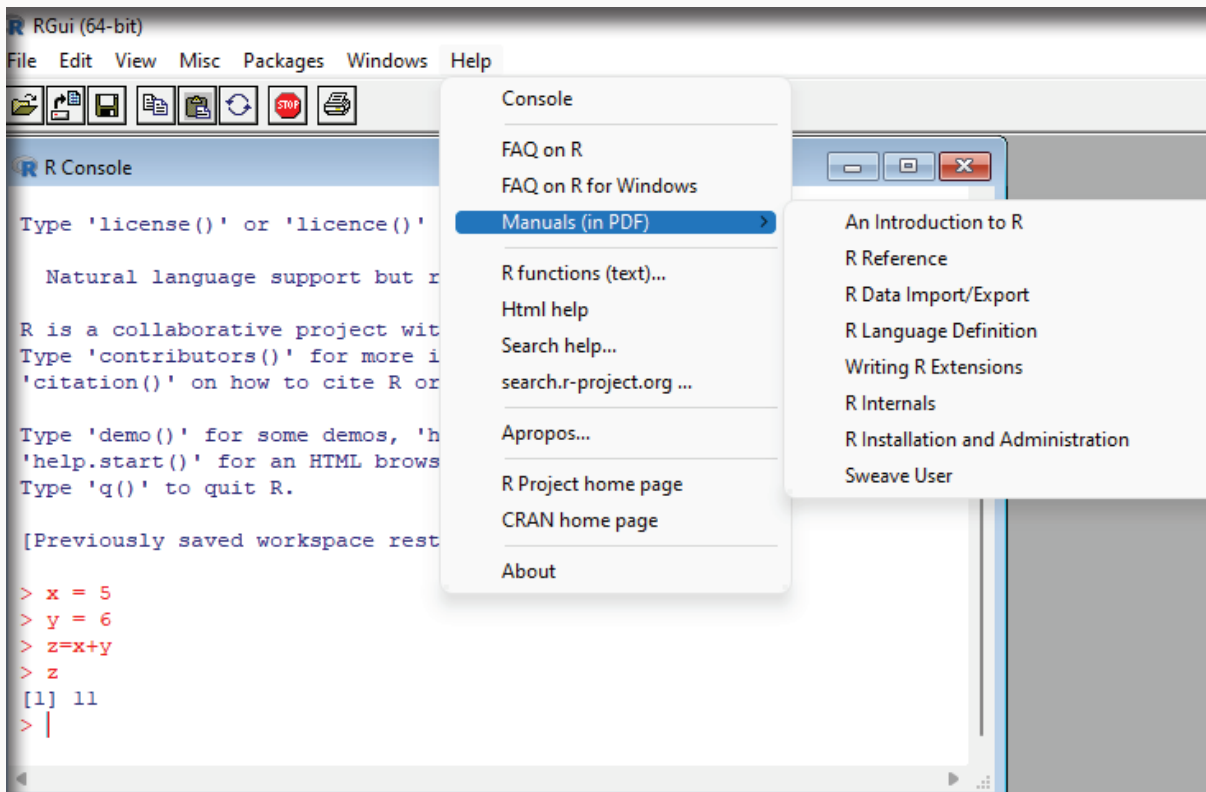


Image showing various manuals listed under Manuals submenu.

R Functions (Text) - This submenu when selected opens up a search box where the user can key in the desired function and search for help.

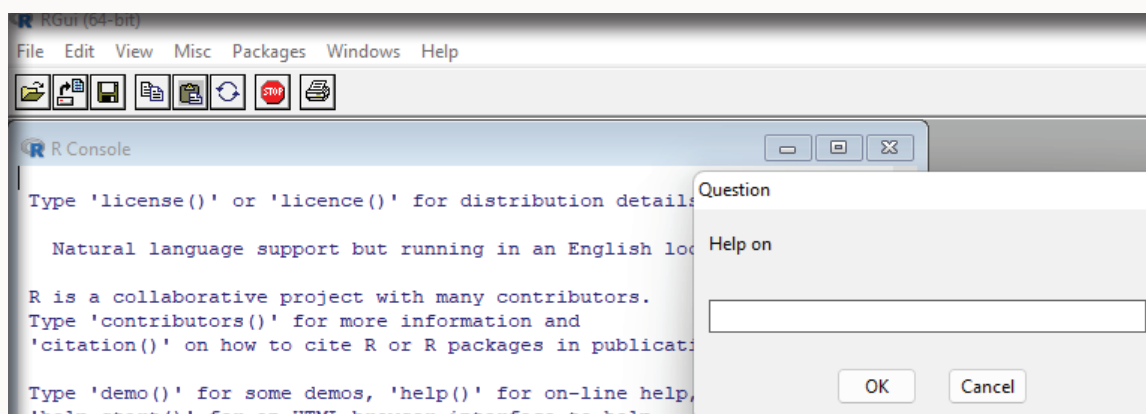


Image showing R Functions (Text) menu

HTML Help - This submenu when clicked displays to the user help files in HTML format.

Search Help - This submenu helps the user to search for relevant help files pertaining to the use of R software.

Search r-Project-.org - This submenu helps the user to look for resources in r project.org webpage.

Apropos - This submenu is a function in R that is used to return a character vector with the names of the objects matching or containing the input character partially.

The image shows a screenshot of the R Console window. The window has a title bar that says "R Console" and standard Windows window controls (minimize, maximize, close). The console displays the following R code and its output:

```
> x = 5
> y = 6
> z=x+y
> z
[1] 11
> help.start()
starting httpd help server ... done
If nothing happens, you should open
'http://127.0.0.1:24463/doc/html/index.html' yourself
> apropos("function")
[1] ".untracedFunction"      "activeBindingFunction"
[3] "all.equal.function"     "as.function"
[5] "as.function.default"    "as.list.function"
[7] "existsFunction"         "findFunction"
[9] "function"               "functionBody"
[11] "functionBody<-"         "getFunction"
[13] "is.function"            "plot.function"
[15] "print.function"         "substituteFunctionArgs"
[17] "sys.function"
> |
```

Image displaying results for the key word 'function' keyed into the apropos box.

R-Studio

This is an integrated development environment (IDE) for R. It has a console, syntax-highlighting editor that supports direct code execution, and tools for plotting, history, debugging and workspace.

Features of R Studio IDE:

1. One can access RStudio locally.
2. It has syntax highlighting, code completion and smart indentation features.
3. Content changes can be viewed in real-time with the visual markdown editor.
4. R help is tightly integrated to R Studio.
5. It has interactive debugger to diagnose and fix errors.
6. It also has extensive package development tools.
7. It has dedicated project folders to keep everything organized.

Unique feature of RStudio is that it is tightly integrated with R programming software (base software). It provides the user with full featured IDE experience and nifty GUI. It should be stressed at this point that RStudio should be installed after installing R. Installation of both these softwares have already been covered in previous chapters. Ideally R programming software should be installed before installing RStudio software.

Getting started:

When R studio opens for the first time R will also be launched as well. It will display three boxes. During the coding phase RStudio will have four different windows. If the 4th window is not visible on the first run all the user needs to do is to click on File/New file/Rscript. The interface will add the 4th block. Background color of all these boxes will be white to start with, it can be changed to user's preference if desired. As soon as RStudio opens up the user will be confronted with a lot of different windows, each with some tabs. This could be overwhelming for the first time user. It is easy to get used to it.

Plain text editor - This is like Notepad. "Plain text" means that no fonts, formatting etc as in word processor. Multiple files can open at once and they appear in tabs. All files can be edited using plain text editor. This can also be used as a script editor. This window can be used to write R code. The main advantage of writing code in this window is that it can be saved and the coding process could be continued in subsequent sessions. This is not possible if scripts are keyed into the console window. Scripts can be used in the console window only to run it and see the output.

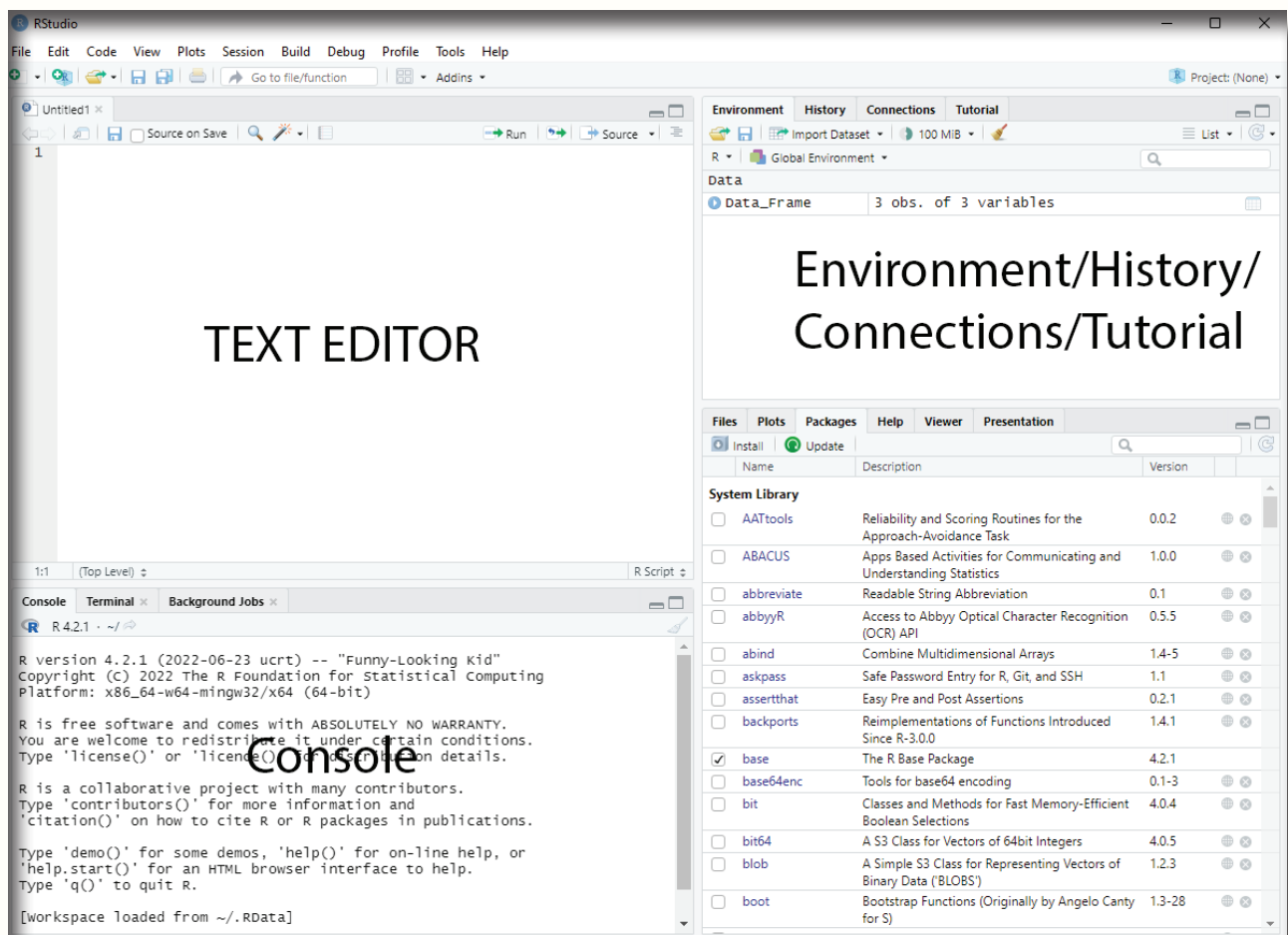


Image showing RStudio interface. Note four compartments. They have been named for convenience by the author.

Default tab in the lower right window is a basic file browser. One can open, delete and rename files there. It is not that well-developed as the operating system's file browser. It is available to help users managing files without switching to other applications that manage files. Rest of the tabs present in this window include (Plots, Packages, Help and viewer).

Packages tab is the next tab seen in the lower right window. This lists out the various installed packages. If the desired package is selected by placing a tick mark in the box in front of the package the same will be loaded into the program.

Plots tab is the third tab seen in this window. When data is formatted in the form of plots the same will be displayed in a window that appears when clicking on this tab.

Help is the next tab. On clicking this tab a window will open displaying help files. User can search for help using this tab.

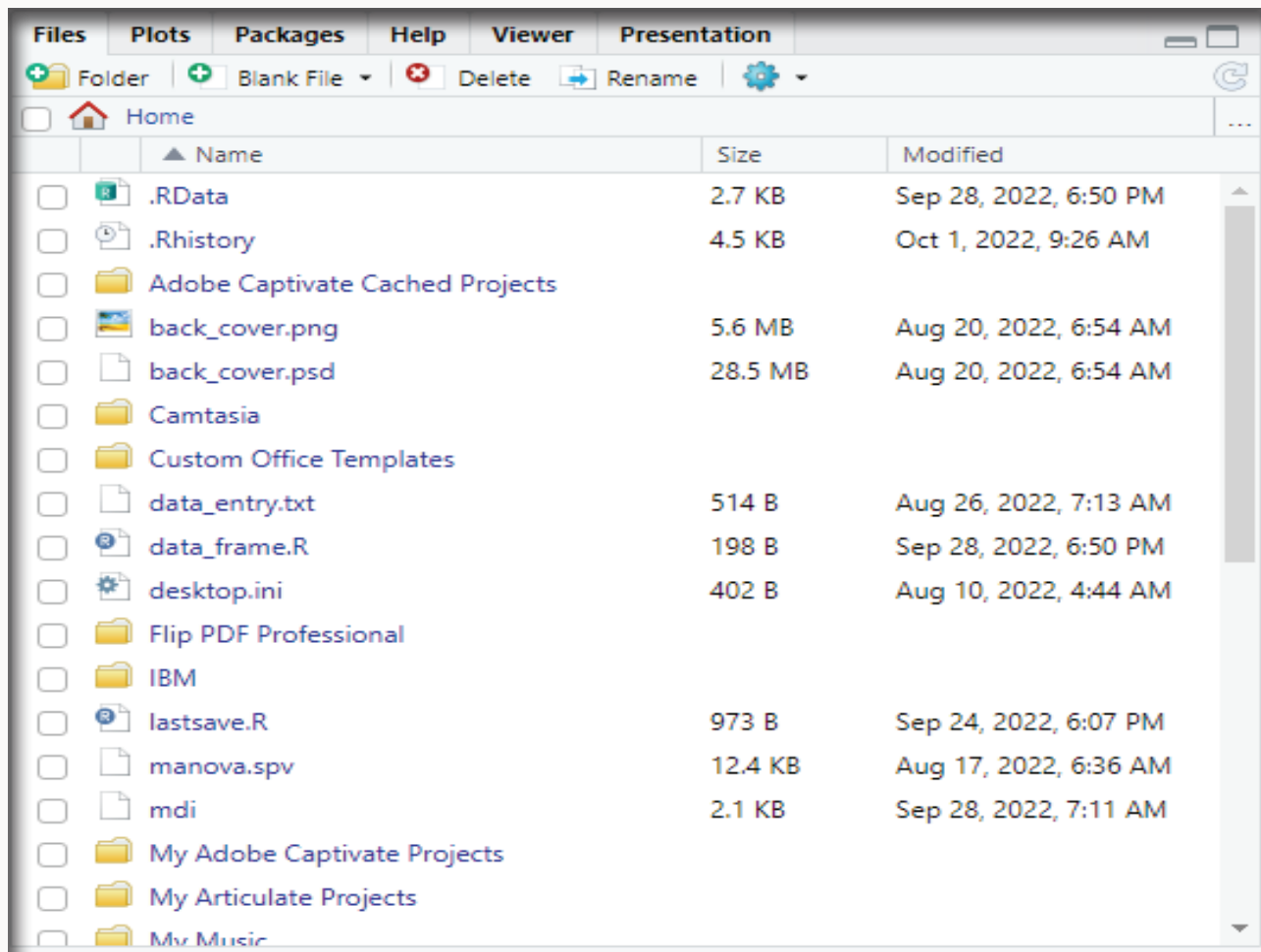


Image showing basic file browser in the lower right window

Viewer - This is the next tab. On clicking this tab a window will open displaying graphs and charts of the data analysed.

Presentation - This is the last tab. RStudio can also be used to create powerful presentations. The created presentations gets displayed in the window that appears when this tab is clicked.

Console:

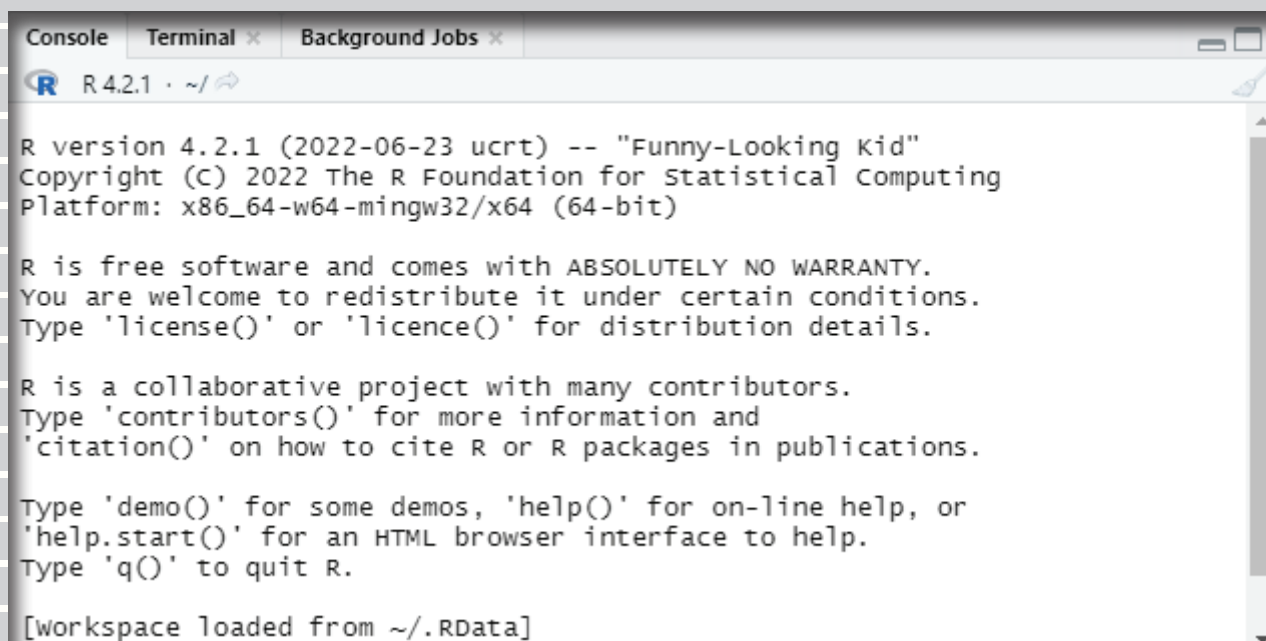
This is a tab in RStudio where the user can run R code. The window pane where the console is located contains three tabs:

Console

Terminal

Jobs

When RStudio is run the console contains information about the version of R the user is working with. Console can be used to test the code immediately. When an expression like $1+3$ is entered one can immediately see the answer output on pressing the Enter key.



```
R 4.2.1 · ~/
```

R version 4.2.1 (2022-06-23 ucrt) -- "Funny-Looking Kid"
Copyright (C) 2022 The R Foundation for Statistical Computing
Platform: x86_64-w64-mingw32/x64 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

[workspace loaded from ~/.RData]

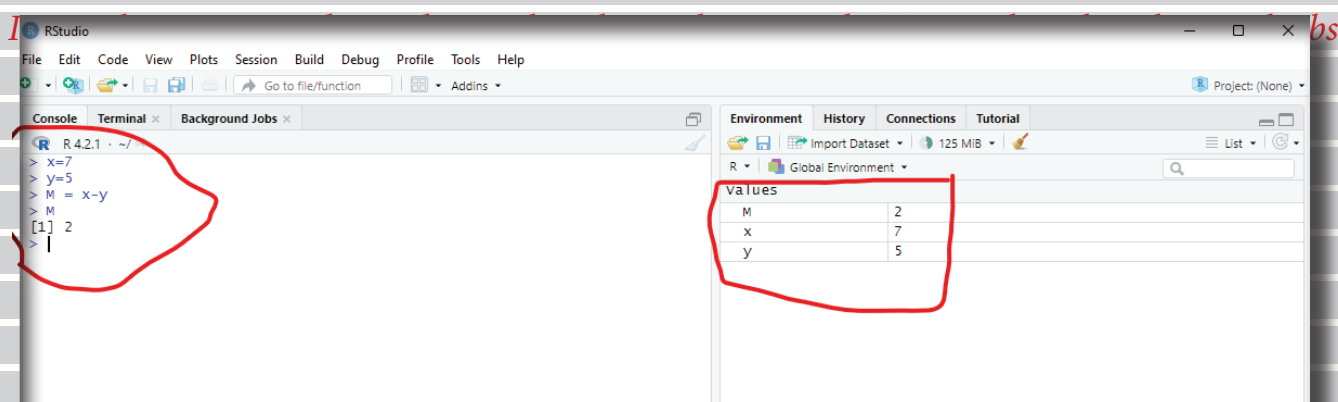


Image showing Console window where code is keyed in. In the environment window the values assigned to each letter (variable) can be seen.

Code entered: > x=7

> y=5

> M = x-y

> M

[1] 2

In the code entered x is assigned a value of 7, while y is assigned a value of 5. M is assigned a value of (x-y). Calculated value of M is 2.

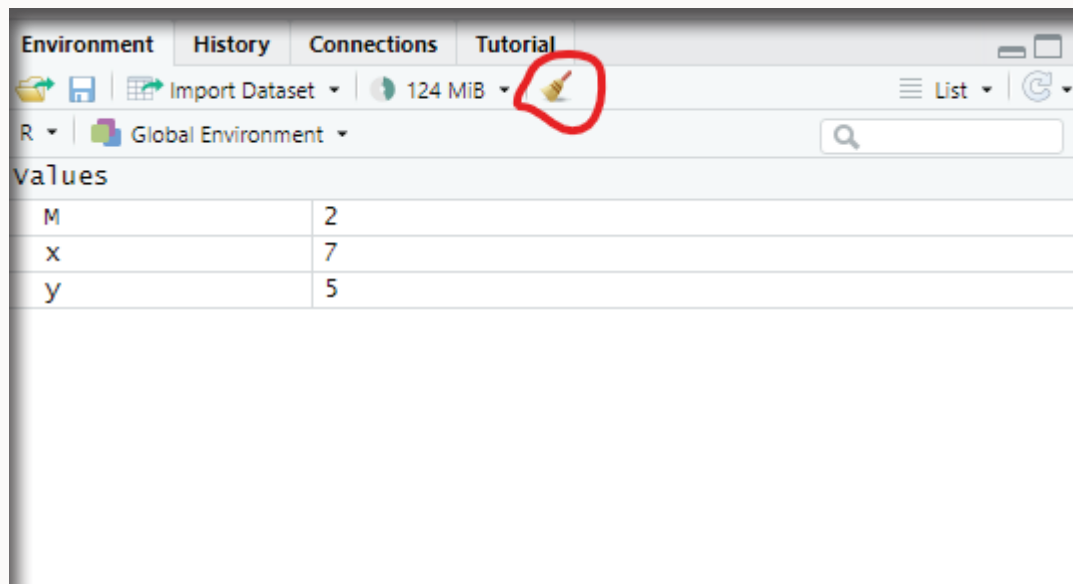


Image showing Environment window where objects are visible. Values of all the three alphabets (variables) can be seen. The window can be cleared of these variables by clicking on the broom icon (red circle).

File menu of RStudio has the following submenu:

New file - This menu allows the user to create new file. It has various submenu which include:

RScript - This will create an environment which can be used by the user to create a new script using R programming.

Quarto document - This is a multi-language, next generation version of R Markdown from RStudio, with many new features and capabilities. Like R Markdown, Quarto uses Knitr to execute R code. This document can include a variety of output types like Executable code block, plots, tabular output from data frames and plain text. To use Quarto with R the user will have to install rmarkdown R package. Installation of packages in R using RStudio will be discussed. This document can be rendered in HTML, PDF or word.

Quarto presentation - Quarto engine can be used for creating presentations in a variety of formats that include:

- revealjs (HTML)
- pptx (PowerPoint)
- beamer Beamer (LaTeX/PDF).

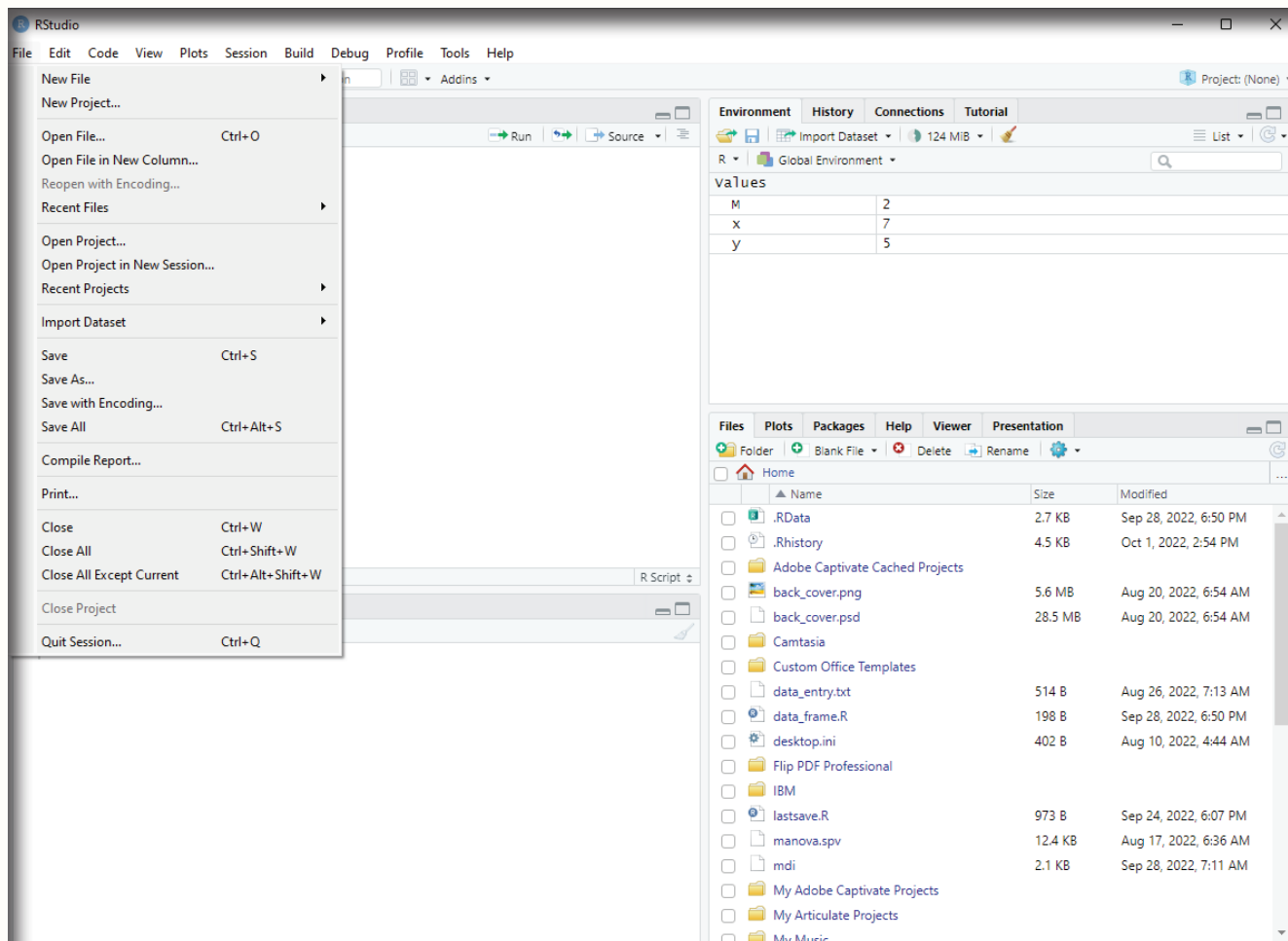


Image showing the File menu of RStudio

R Notebook - This is a R Markdown document that allows for independent and interactive execution of code chunks. It can be considered as a unique execution mode for R Markdown documents and any R Markdown document can be used as a notebook, and all R Notebooks can be rendered to other R Markdown types.

R Markdown - This provides an authoring framework for data science. One can use a single R Markdown file to both

Save and execute code

Generate high quality reports that can be shared.

These documents are fully reproducible and support dozens of static and dynamic output formats.

Shiny Web app - This is a R package that makes it easy to build interactive web apps from R. Using this one can host standalone apps on a webpage or embed them in R Markdown documents or build dashboards.

These applications can be extended using CSS themes, htmlwidgets and JavaScript actions.

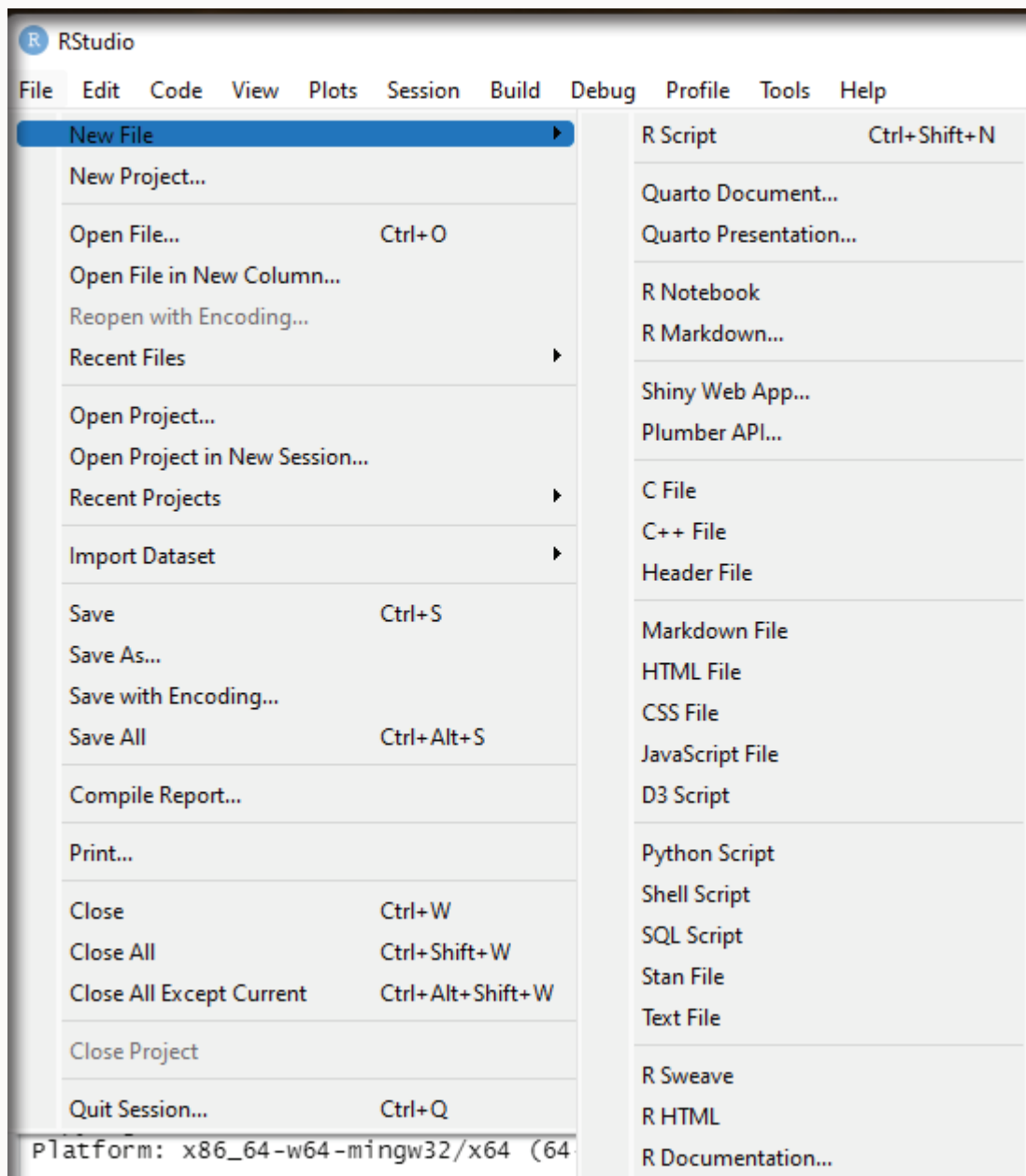


Image showing various submenu listed under New file menu

The user will have to install the Shiny package.

This can be installed by opening an R session and running the following code:

```
install.packages("shiny")
```


Plumber API - This allows the user to create a web API by just decorating the existing R source code with roxygen2 - like comments. These comments allow plumber to make the R functions available as API end-points.

C file - R programming tool can be used to create C code. In order to compile c/C++ code R requires installation of additional build tools.

C++ file - R programming tool can be used to create C++ code. R needs to install some additional build tools for this function.

Header files - This can be used together with raster binary files to read data in other applications. Some additional C libraries need to be installed for creation of this file.

Markdown file - This menu can be used to create R Mark down file. Markdown is a simple formatting syntax for authoring HTML, PDF, and MS word documents.

HTML file - Using this submenu the user can create a HTML file.

R Programming software can also be used to create:

- Javascript
- D3 script
- Python script
- Shell script
- SQL script
- Stan file
- Text file.

These scripts and files can be created by clicking on the relevant submenu listed under New submenu under File menu.

R Sweave - This is a function in the statistical programming language R that enables integration of R code into LaTeX documents. The main purpose of this feature is to create dynamic reports that can be automatically updated if data or analysis changes. Sweave document can be created by clicking on the submenu R Sweave listed under New submenu.

R HTML - R Programming can be used to create HTML files with R code embedded in it. This is known as R HTML. The user can invoke this feature by clicking on the R HTML submenu.

R Documentation - Document that is prepared using the features available in R. The file goes under the term R Documentation. User who prefer to create document in R Document format can click this submenu and a template will be displayed. The document can be created following the displayed template.

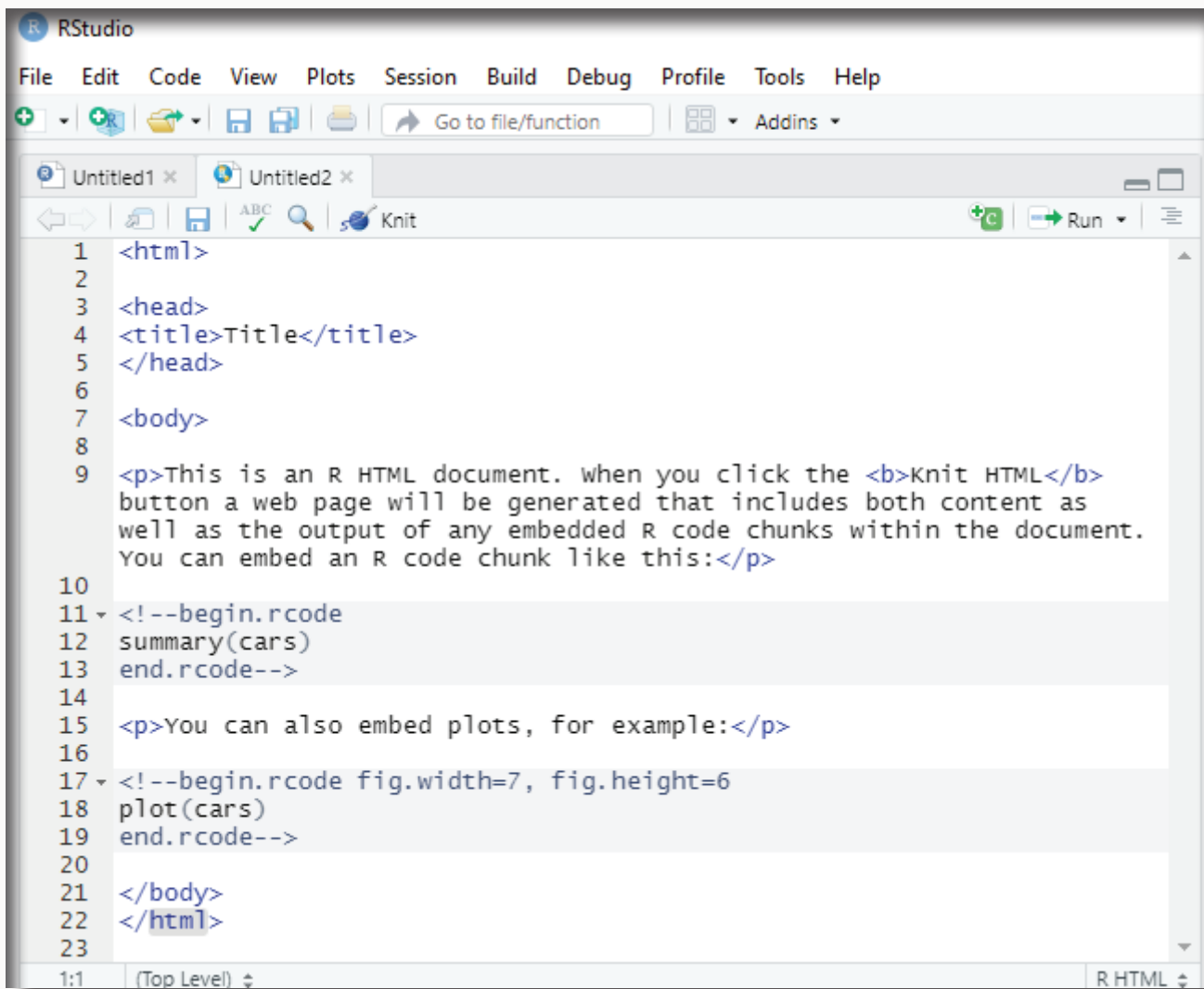


Image showing R HTML template document which gets displayed when the R HTML submenu is clicked

Edit Menu - This menu that is available at the top of R Studio window can be used to perform various edit functions. The submenu available under this menu include:

Back

Forward

Undo

Redo

Cut

Copy

Paste

Paste with Indent - This submenu allows the user to get correct indentation while pasting the R code.

Folding - Has 4 subemnus under it. The source pane in RStudio IDE supports both automatic and user-defined folding of regions of code. Code folding allows the user to easily show and hide code blocks to make it easier to navigate the source file and focus on the coding task at hand.

Foldable regions:

The following types of code regions are automatically foldable within RStudio:

- . Braced regions
 - . Code chunks within R Sweave or R Markdown documents
 - . Text sections between headers wtihin R Markdown documents
 - . Code sections
1. Collapse
 2. Expand
 3. Collapse All
 4. Expand All

Go to Line

Find

Find Next

Find Previous

Use Selection for find

Replace and Find

Find in File

Check spelling

Word count

Clear Console

Majority of these submenu are self explanatory.

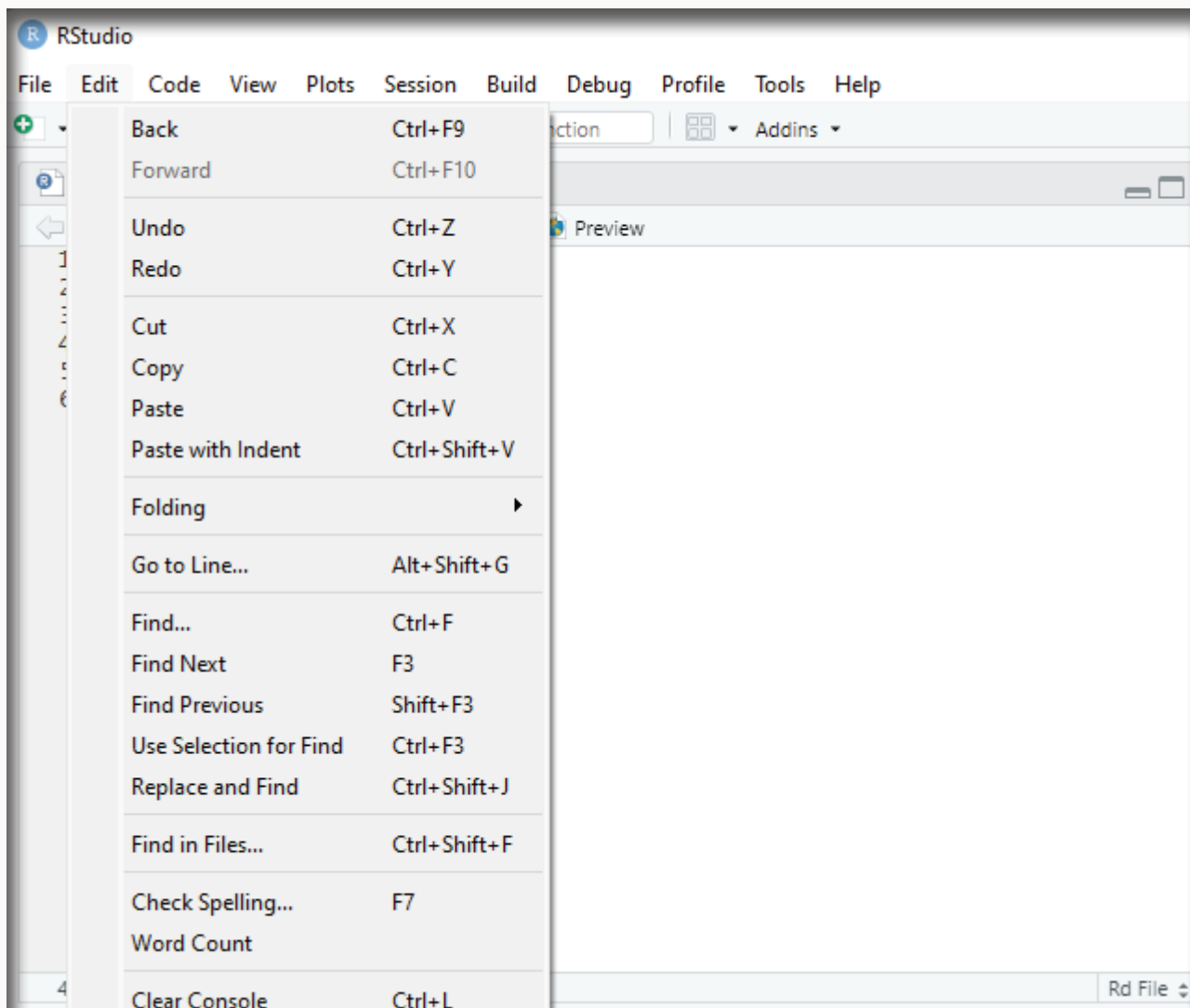


Image showing submenu listed under Edit menu

Code Menu:

Code menu contains the following Submenu:

Go To File/Function

Soft Wrap Long Lines - Enabled by default

Rainbow Parentheses - This setting will replace other types of brackets

Terminal

Source File

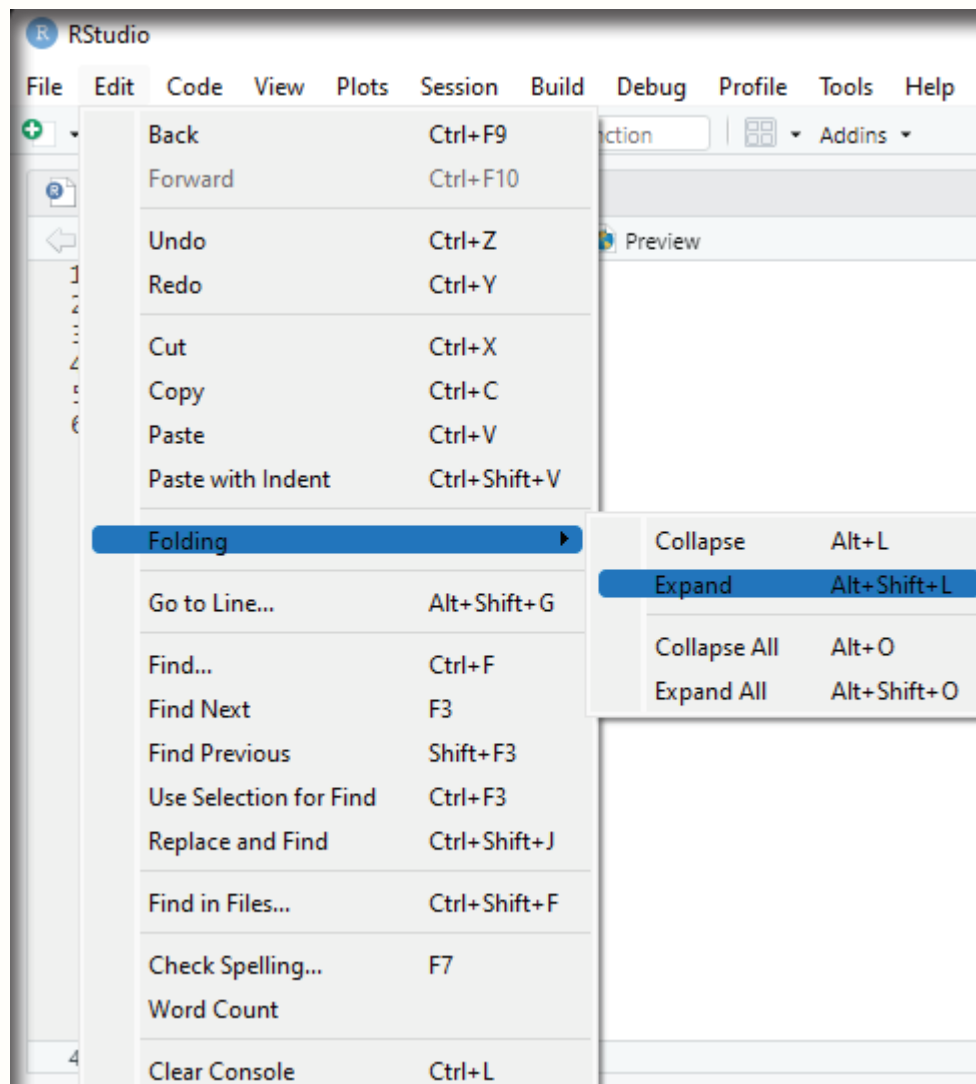


Image showing Submenu under Folding menu

Code Menu also provides submenu that can be clicked to run the code either fully or from a selected point.

A code from the code window can be selected and their exact function can be extracted using Extract Function Submenu. Similarly variables from the selected code can also be extracted using Extract Variables submenu.

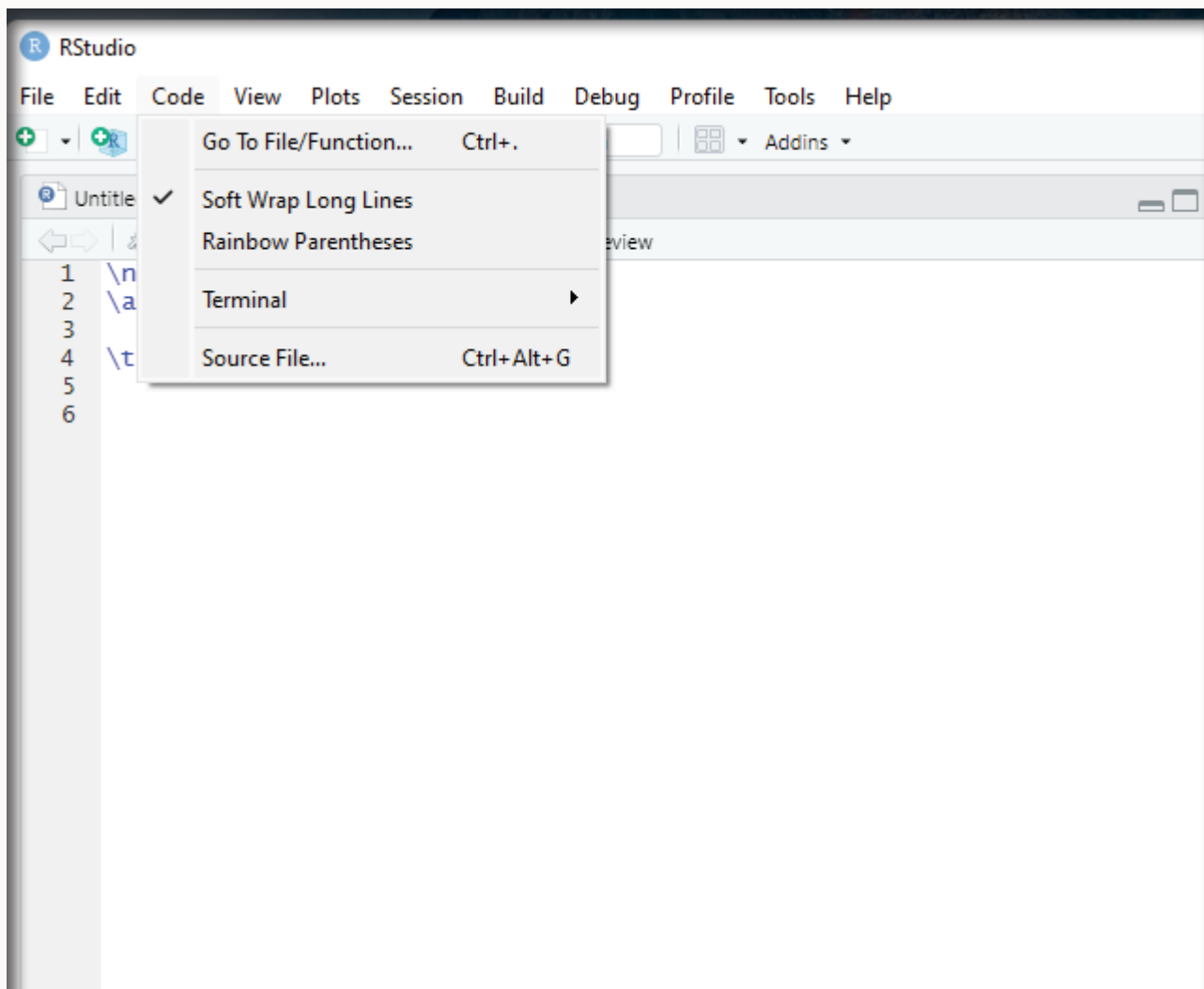


Image showing Code Menu and its various submenu

View menu can be used to hide / unhide tool bar.

Tweak location and number of Panes

Tweak the size of the window by zooming in and out

Switch to a specific tab

Move focus to source

Move focus to console

Move focus to terminal

Move focus to help

Show files

Show plots

Show viewer

Show environment

Show presentation

Show connections

Show tutorial

Show background jobs

Show other panes

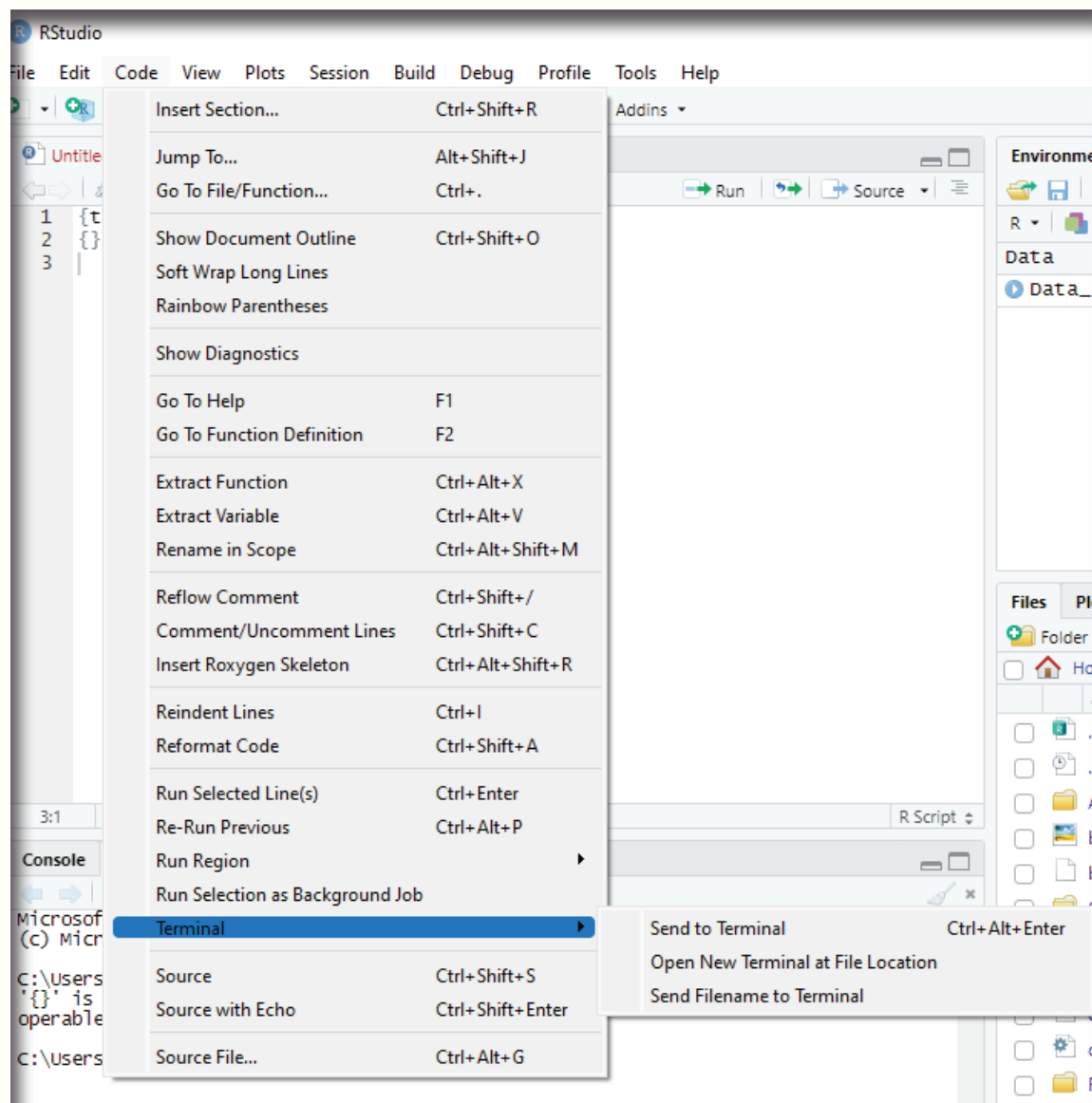


Image showing Terminal and its submenu

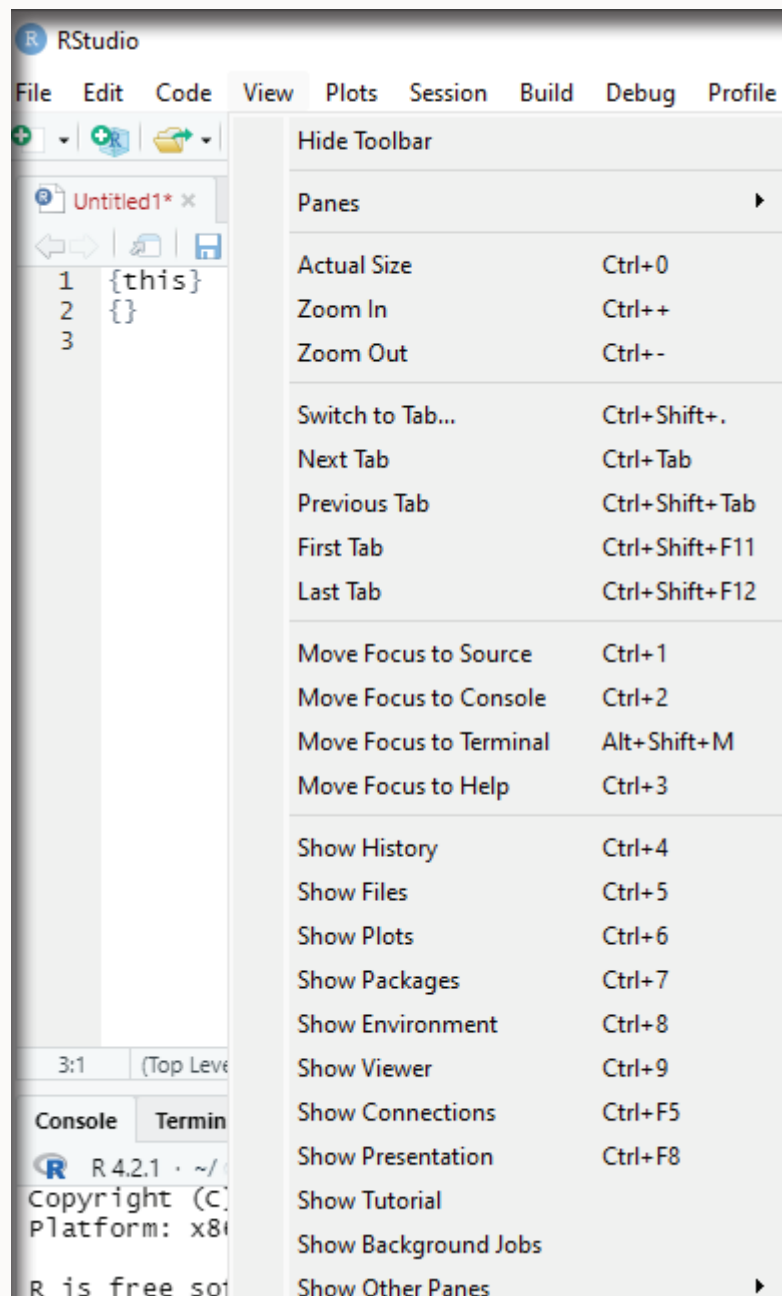


Image showing View Menu and its submenu

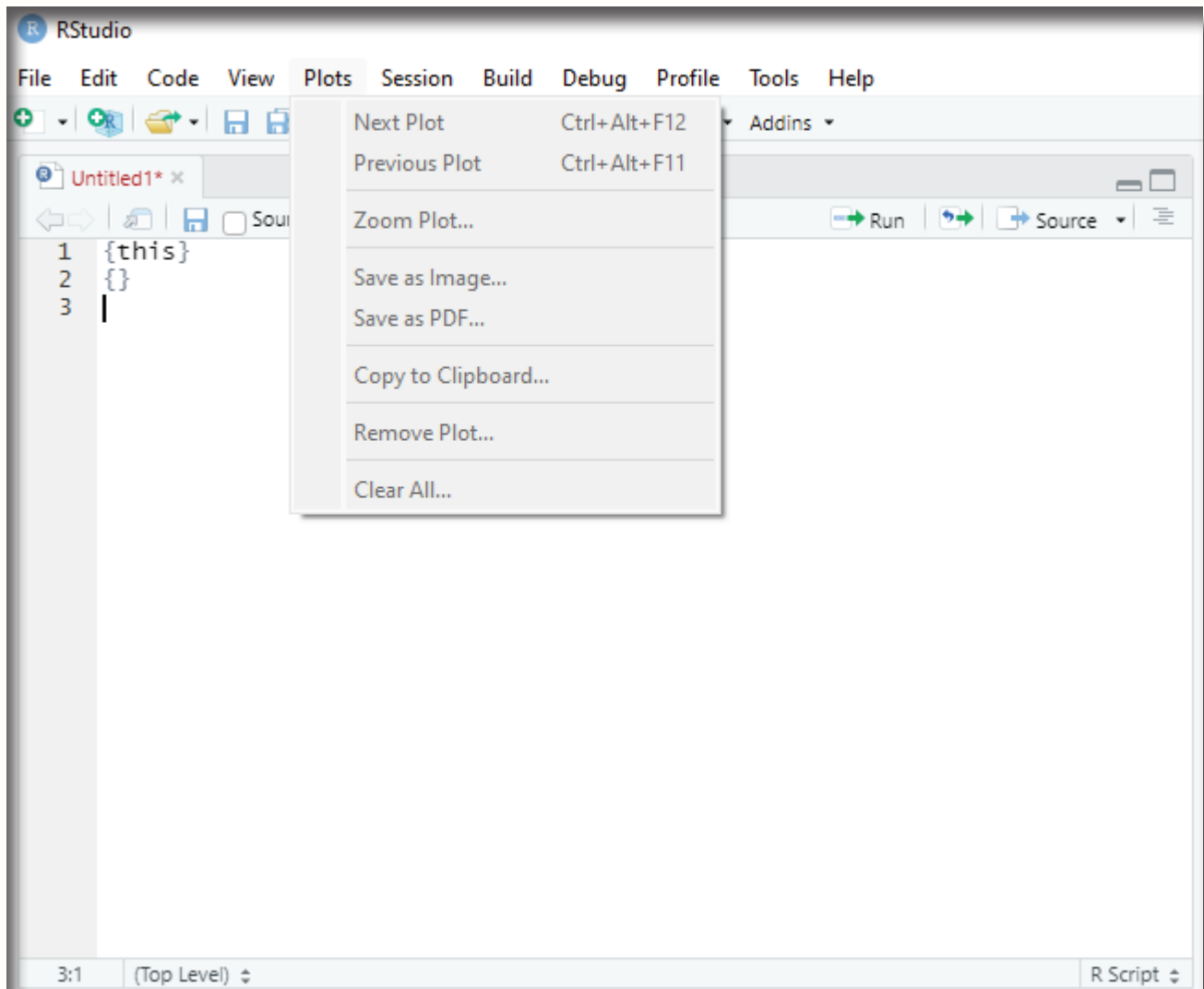


Image showing Plots menu and its submenu

Plots menu - Plots menu can be used to migrate to various plots held in the RStudio.

Debug Menu - This menu is used to debug R code that has been keyed into the console. It runs the code line by line and displays error code thereby helping the user to troubleshoot code errors.

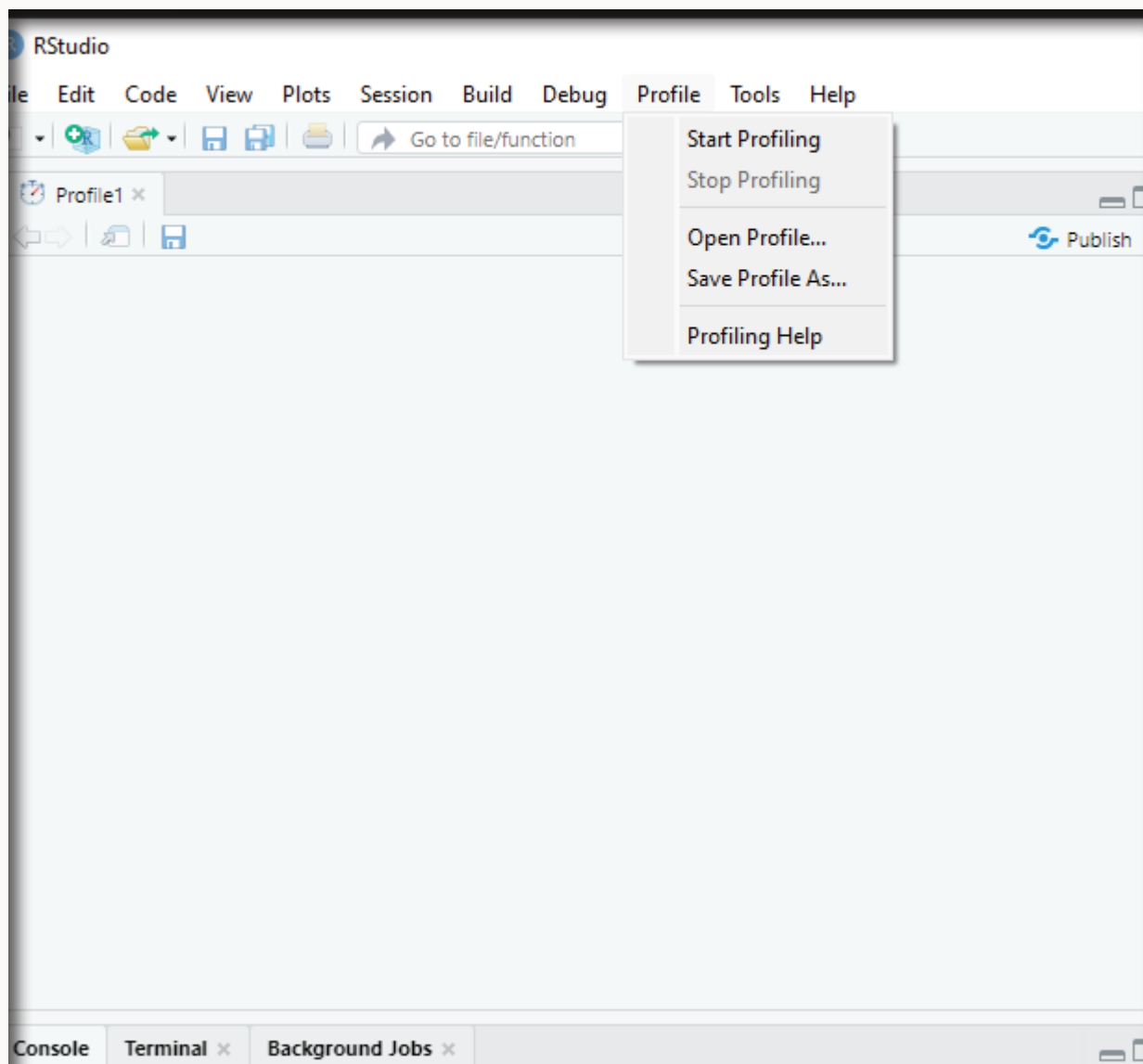


Image showing Profile menu and its submenu

Profiler is a tool that helps the user to understand how R spends its time. It provides an interactive graphical interface for visualizing data from Rprof. This is R's built in tool for collecting profiling data. Profiler can be run by choosing start profiling submenu from the Menu Profile. The same can be stopped by clicking on Stop Profiling submenu. Help files pertaining to profiling can be accessed by clicking on Profiling Help submenu from Profile menu.

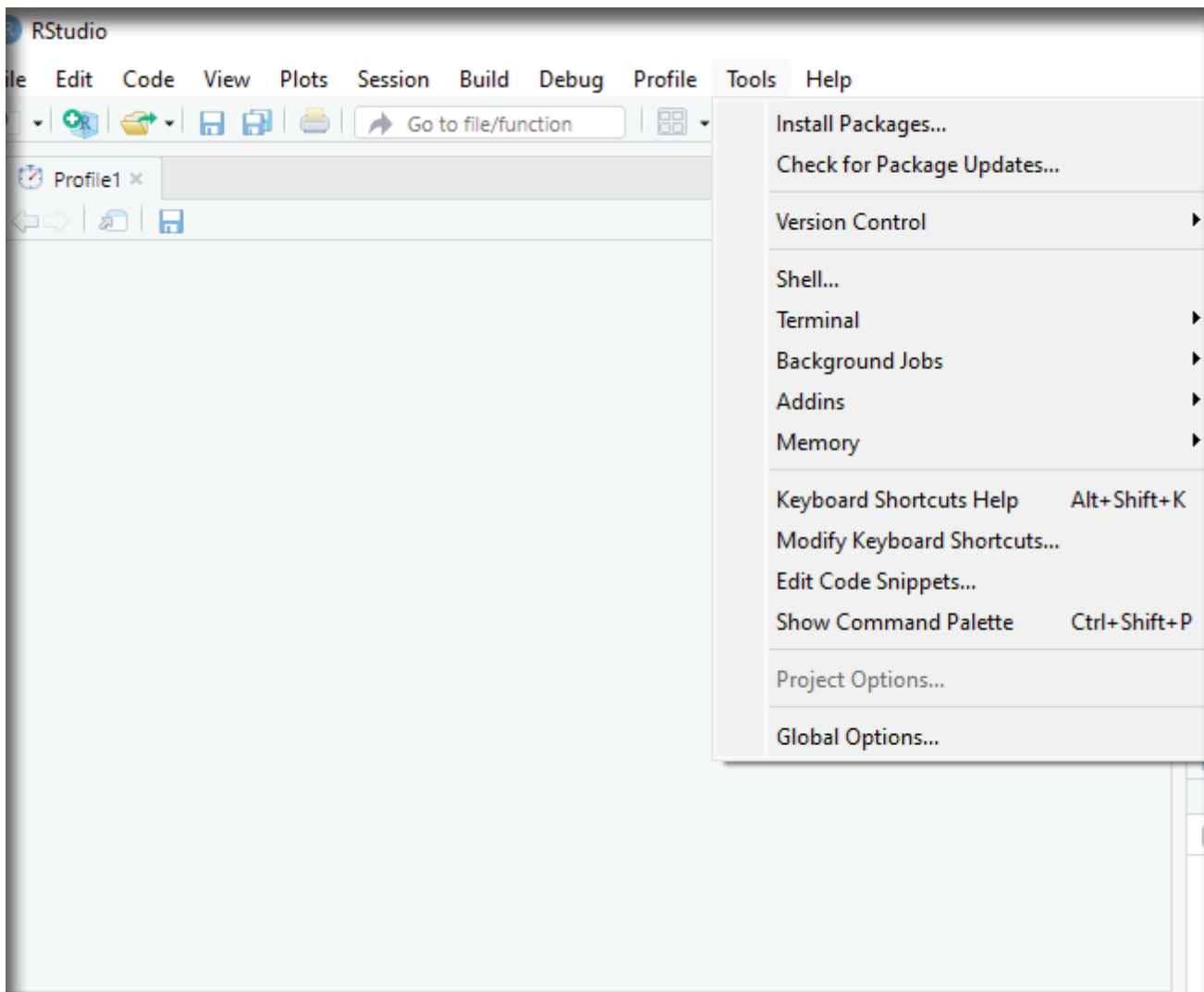


Image showing Tool menu and its submenu

Tools menu:

The following are submenus listed under this menu.

Install Packages - This submenu can be used to install R packages.

Check for Package updates - This submenu can be used to check and install updates for already installed packages if available.

Version control - This submenu helps software teams using RStudio software teams to manage changes to source code over time. Version control software keeps track of every modification to the code in a special kind of database. If a mistake is made, developers can turn back the clock and compare earlier versions of the code to help fix the mistake while minimizing disruption to all team members. To make use of this feature the user needs to decide on which control system to use. It can be Git or Subversion. Both of these systems are supported by R. Git or subversion should be installed into the operating system and RStudio restarted for this version control system to work. Version control can be invoked only from a project setup.

Shell - This is also known as a bash or terminal. This program can be used to run other programs, rather than do calculations itself. In windows this menu will open up windows command prompt.

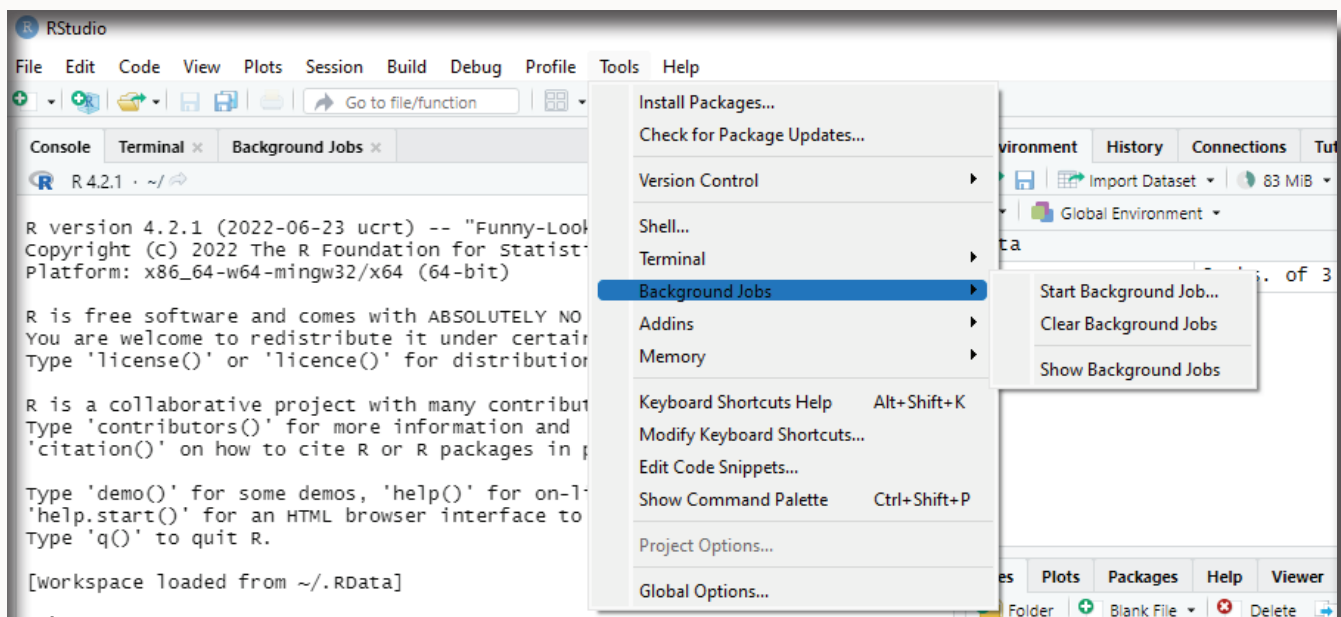


Image showing Background jobs submenu and its various submenus.

Background jobs - RStudio has the ability to send long running R scripts to local and remote background jobs. This functionality can improve the productivity of data scientists and analysts using R since they can continue working in RStudio while jobs are running in the background. Running a Shiny application as a local background job allows the current R session to remain free to work on other things. Three submenus are available under this menu which include:

Start background job - Can be used to start a new background job.

Clear background job - Can be used to clear background jobs

View background job - Can be viewed to see running background jobs.

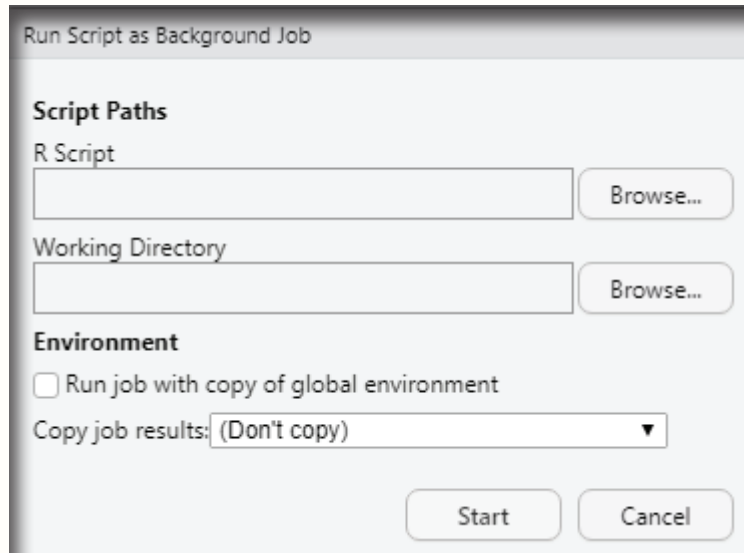


Image showing the dialog box that prompts the user to locate the script file that needs to be run in the background. On showing the path to the script file and clicking on the start button the script will be run in the background.

Terminal :

Terminal in RStudio provides access to the system shell within the RStudio IDE. Uses of Terminal window includes, advanced source control operations, execution of long running jobs, remote logins, and interactive full screen terminal applications (text editors, terminal multiplexers).

Submenu under terminal include:

New terminal - If this submenu is clicked new terminal window will open.

Go to Current Directory - clicking on this submenu takes the user to the current working directory.

Rename Terminal - User can open more than one terminal window for working. When more than one window it would cause confusion to the user. This submenu provides flexibility to the user to rename the Terminal. By default Terminal window will be suffixed by numbers like (1,2, 3 etc.,) To avoid confusion if more than one terminal is created by the user then it should be renamed.

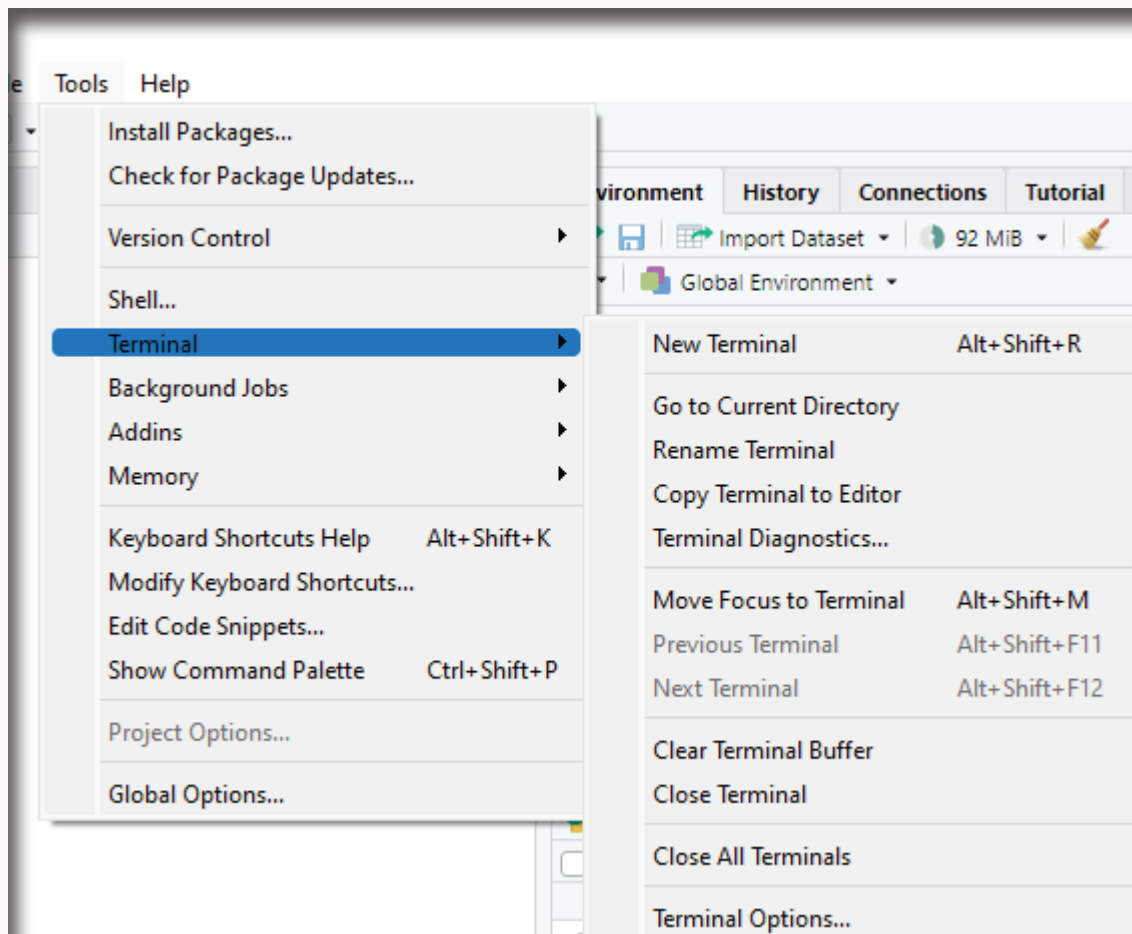


Image showing Terminal submenu along with its submenu

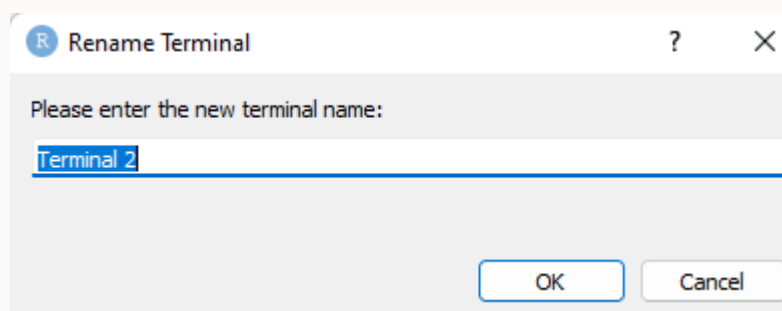


Image showing Rename Terminal window

Copy Terminal to Editor - Contents of the terminal can be directly copied to the Editor window by clicking on this submenu.

Terminal Diagnostics - This submenu can be used to retrieve details about Terminal windows. It provides details about number of terminal windows open etc., It also provides information about the system.

Move Focus to Terminal - This submenu on being clicked moves focus to the terminal window.

Previous Terminal - Clicking on this submenu will open the previous terminal window if there are more than one terminal opened up.

Next Terminal - Clicking on this submenu will take the user to the next terminal.

Clear Terminal Buffer - This submenu will clear the contents of the terminal.

Close Terminal - This submenu closes the Terminal window that is in focus.

Close All Terminals - This submenu when clicked closes all the terminals created.

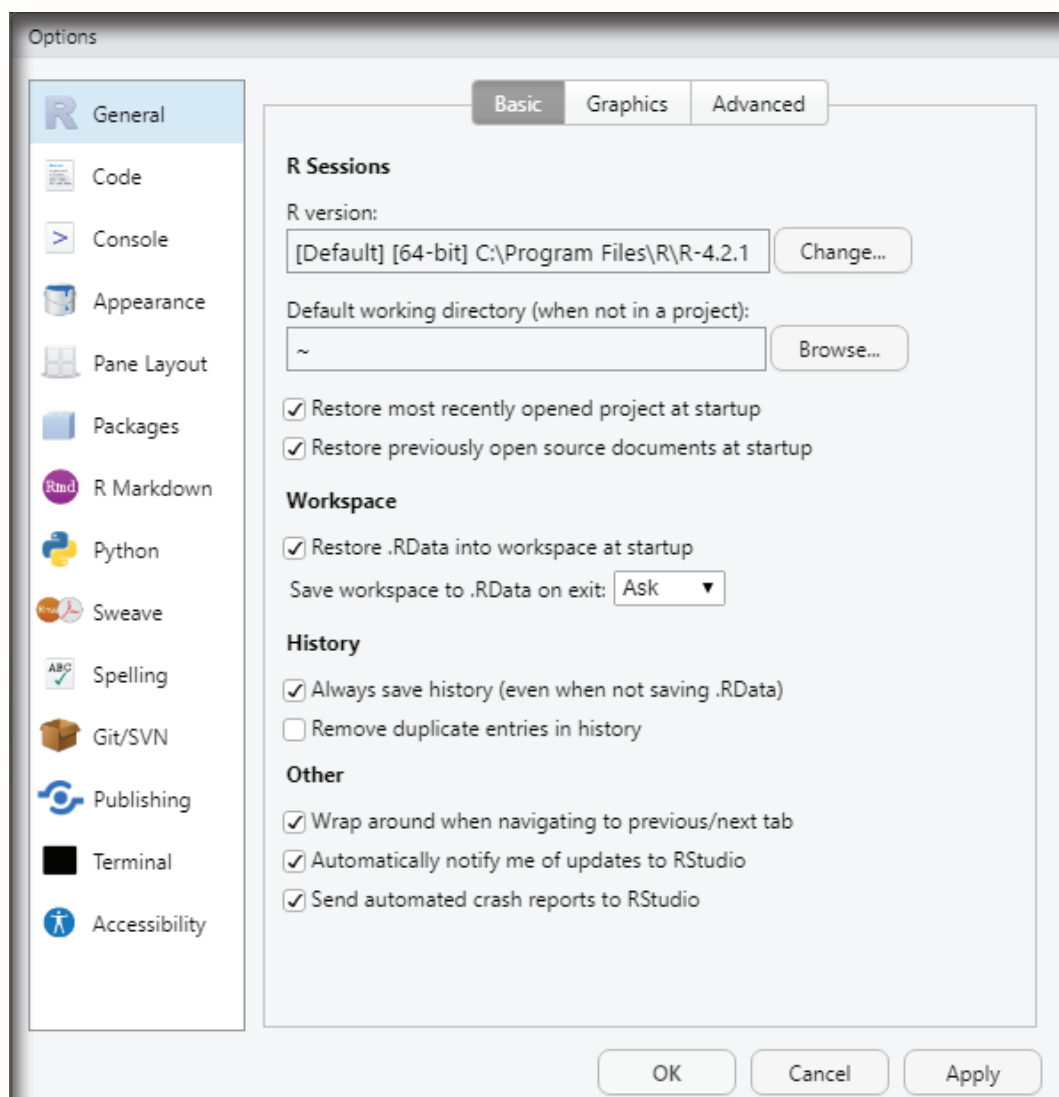


Image showing Terminal options window

Global Terminal Information

Loaded TerminalSessions: 2

Handle: '6F1B2D42' Caption: 'Terminal 1'

Handle: 'EAFDFBAC' Caption: 'Terminal 2'

Terminal List Count: 2

Handle: '6F1B2D42' Caption: 'Terminal 1' Session Created: true

Handle: 'EAFDFBAC' Caption: 'Terminal 2' Session Created: true

Global Terminal Information

Caption: 'Terminal 2'

Title: ''

Cols x Rows '87 x 21'

Shell: 'Command Prompt'

Handle: 'EAFDFBAC'

Sequence: '2'

Restarted: 'false'

Exit Code: 'null'

Full screen: 'client=false/server=false'

Zombie: 'false'

Track Env 'false'

Local-echo: 'false'

Working Dir: 'Default'

Interactive: 'Always'

WebSockets: 'true'

System Information-----

Desktop: 'true'

Remote: 'false'

Platform: 'Windows'

Connection Information

2022/10/4 14:50:34: Connect WebSocket: 'ws://127.0.0.1:5950/terminal/EAFDFBAC/'

2022/10/4 14:50:34: WebSocket connected

Local-echo Match Failures

<Not applicable>

Image showing Terminal Diagnostics window

Terminal Options: This submenu will open up window where Options pertaining to terminal can be set.

Keyboard shortcut help - This submenu on being clicked opens up keyboard shortcut help. This opens up a window showing various keyboard shortcuts that can be used in RStudio.

Modify keyboard shortcuts - Using this submenu the default keyboard shortcuts can be modified.

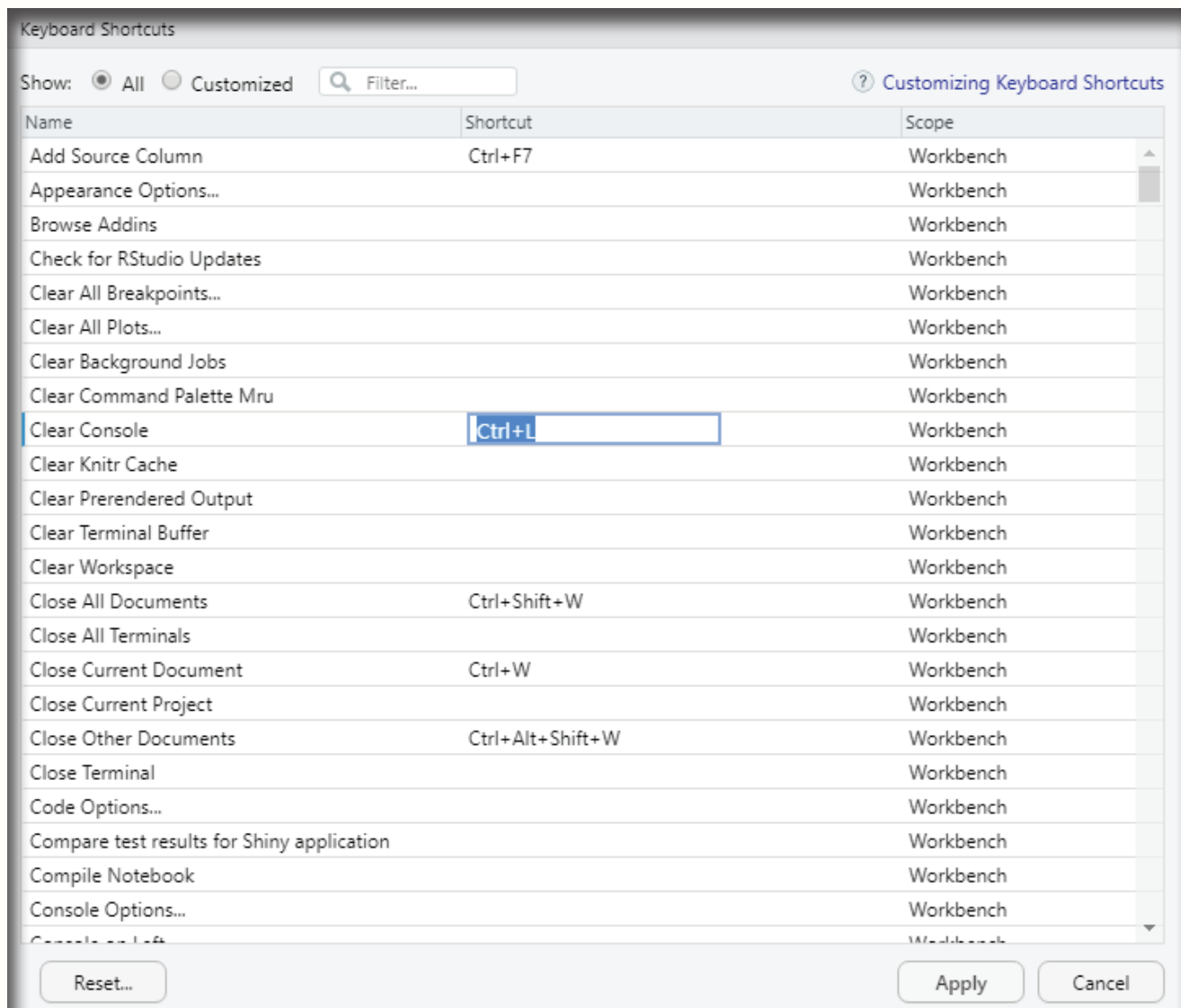


Image showing keyboard shortcut change window

Edit Code Snippets - Code snippets are text macros that are used for quickly inserting common code snippets. If a snippet is selected from the completion list it will be inserted along with several text placeholders which can be filled by the user by typing and then pressing tab to advance to the next placeholder. The pre-saved code snippets can be edited by the user using this submenu.

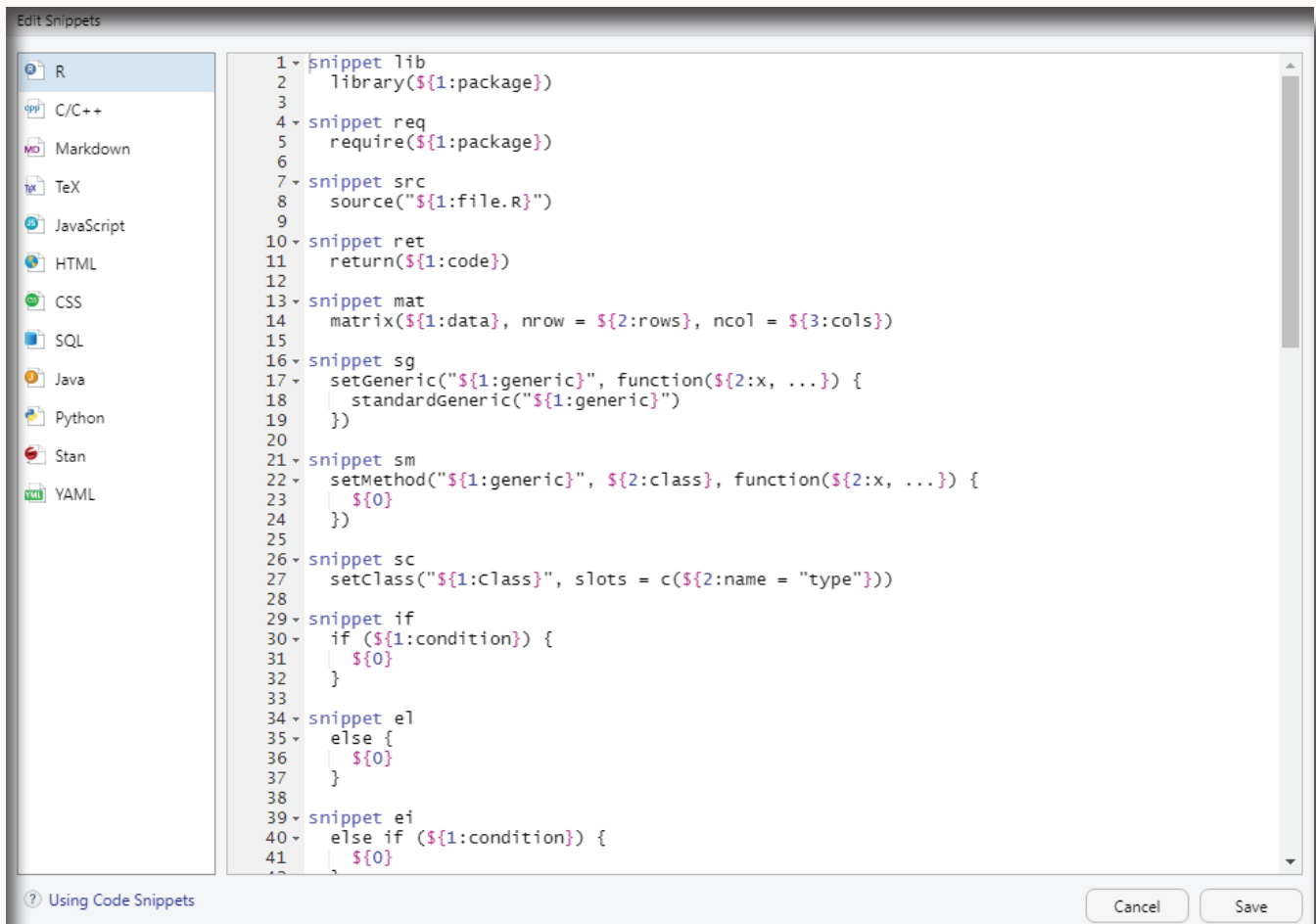


Image showing Code snippet edit window where code snippets can be edited

Global options submenu - This submenu on being clicked opens up Global options window where various settings of RStudio can be changed.

Help:

This menu provides all help files of RStudio under one menu for the benefit of the user.

Data An Introduction

Types of Data in R

In any programming language the user needs to use various variables to store various information. Variables are nothing but reserved areas in memory locations to store values. When one creates a variable, some space is reserved in the memory module.

The user can store information of various data types like character, wide character, integer, floating point, double floating point, Boolean etc.

Character - Includes letters, numerical digits, common punctuation marks and whitespace.

Wide character - is a character datatype that is generally greater than the traditional 8-bit character.

Integer - Is a numerical data.

Floating point - This is a positive or negative whole number with a decimal point.

Double floating point - This is a number format occupying 64 bits in computer memory. A double floating point can hold up to 15 digits.

Boolean - This is actually a true or false data. This is a system of logical thought that is used to create true/false statements. This is also known as Logical type of data.

In R, there are 6 basic data types:

Logical - Logical data type in R is also known as boolean data type. It can have two values: TRUE and FALSE (all upper case).

Numeric - In R, the numeric data type represents all real numbers with or without decimal values.

Integer - The integer data type specifies real values without decimal points. If suffix L is used it specifies integer data. (186L)

Complex - The complex data type is used to specify purely imaginary values in R. One can use the suffix i to specify the imaginary part.

Character - The character data type is used to specify character or string values in a variable. For example "A" is a single character and "APPLE" is a string. One can use ' or "" to represent strings.

Raw - A raw data type specifies values as raw bytes. The user can use the following method to convert character data types to a raw data type and vice-versa:

`charToRaw()` - Converts character to raw data

`rawToChar()` - Converts raw data to character data.

There are basically 5 different data objects in R that are commonly used. They include:

1. Vector
2. Matrix
3. Array
4. Lists
5. Data Frames

In contrast to other programming languages the variables in R are not declared as some data type. The variables are assigned with R-Objects and the data type of the R-Object becomes the data type of the variable. Frequently used types of R-Objects include:

Vectors - Vector is a basic data structure which plays an important role in R programming. In R, a sequence of elements which share the same data type is known as a vector. A vector supports logical, integer, double, character, complex or raw data type. Elements contained in vector are known as components of the vector. The user can check the type of vector with the help of `typeof()` function. Length is an important property of a vector. A vector length is basically the number of elements in the vector, and is calculated with the help of `length()` function.

Simply stated a Vector is a sequence of data elements of the same basic type.

There are 5 classes of vectors also termed as Atomic Vectors:

- * Logical - This type of vector can either take a value of TRUE or FALSE. (Note all these letters should be in upper case).
- * Integer - Takes a whole number value. Example (15L, 30L, 4566L). R is capable of handling integers that are fairly long i.e., 32-bit long. Hence, L is used as a suffix after the integer to indicate to R that it is a long integer.
- * Numeric - Can take a whole number or a decimal number. Example (6, 4.876).
- * Complex - R support complex data types that are a set of complex numbers. The complex data type is to store numbers with an imaginary component and hence is suffixed with an 'i'.

* Character - Single character of a sequence of characters forming a word. This data type should be entered between " of "" to indicate to the software that the data type is character.

Example 'A' "Hello".

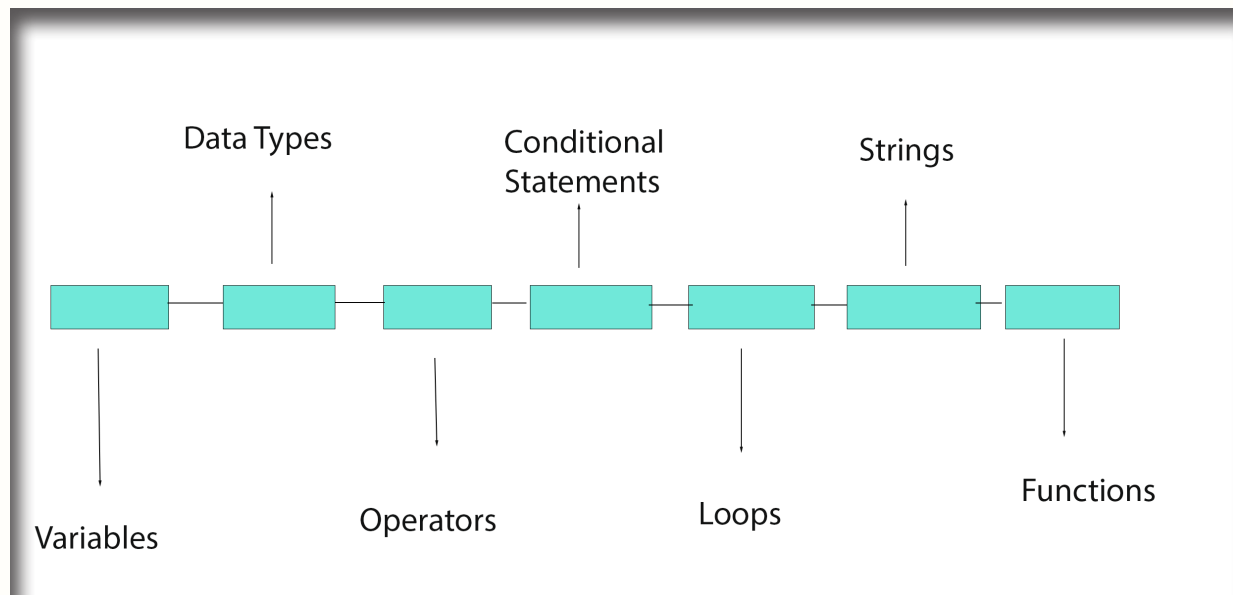


Image showing the heirarchy of R Programming in data analysis

What are variables?

Variable is a reserved memory locations to store values. When the uesr creates a variable some space is reserved in the memory. In lay terms it can be compared to a container that can hold only one material. The type of material can vary. The software identifies the type of data that has been allocated to a variable and allots a suitable memory place to hold on to it. This data is held on to the memory till such time when the user replaces it with another data.

Data types:

This helps in classification of the type of data that is held in a variable. The class or type of the data held in the memory allocated to the variable is important because the size of the memory block allocated varies according to the type of the data contained in the variable. Classification of data type held in the variable is important because it helps in the user in performing different types of opeartions using R Programming language. For example if the data type happens to be numeric then arithmetic calculations, logical operations and string operations can be performed using R programming software. These same operations cannot be performed if the variable holds a character data. When one considers vector as a whole, either one can have a single element belonging to one of the above described data types or it can be a sequence of elements.

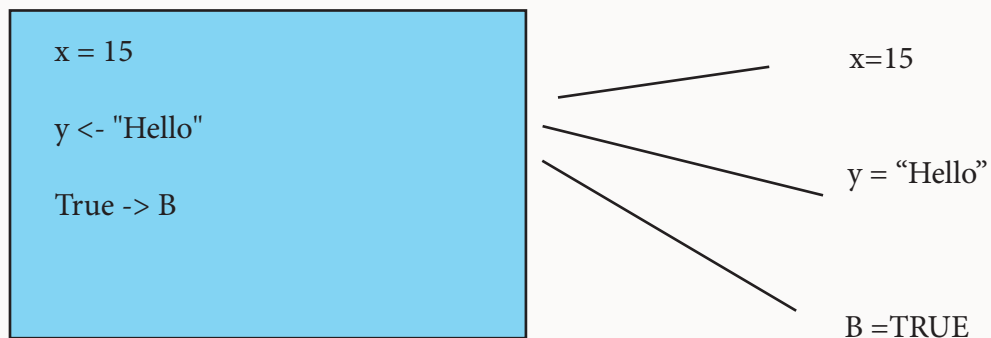


Image showing three variables and their values coded. Variable x has been assigned a value of 15, numeric variable. variable y has been assigned a value of "Hello" a character variable and Variable B has been assigned a boolean value of TRUE.



Image showing 5 different data types used in R Programming

In order to demonstrate the various data types in R, one has to open the R studio. The scripting area should be used to key in the scripts. This is a must when the user needs to write multiple lines of code. The console area can be used to execute a single line of code. Every time the user declares a variable it gets automatically updated in the Global Environment window.

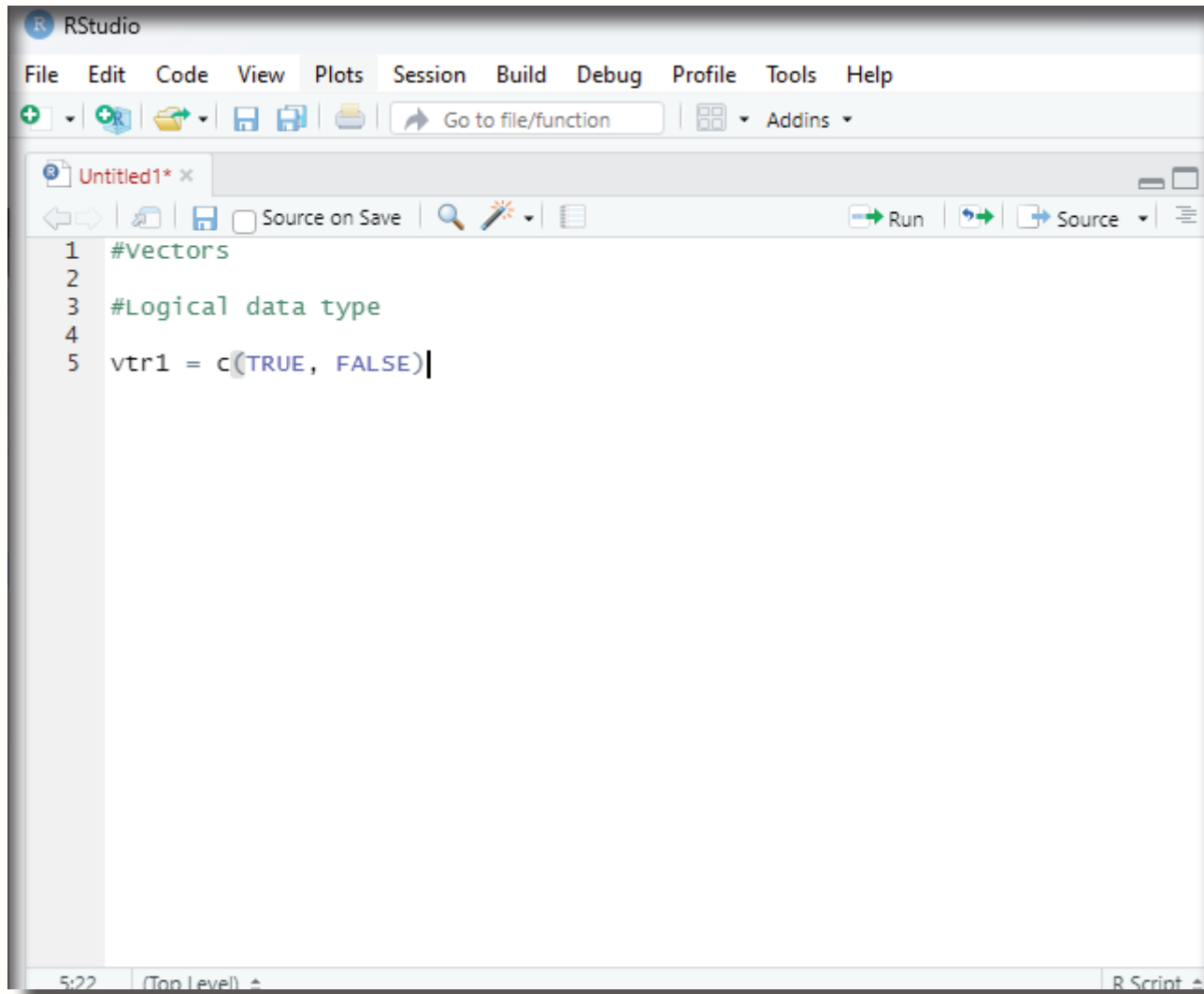


Image showing code entered into scripting window. After entering the code it can be run on clicking the Run button. The output will be displayed in the console window.

#Vectors

#Logical data type

vtr1 = c(TRUE, FALSE)

Note the code block above. This code block can be used to allocate variables to a vector. In this code name of the vector is given as vtr1 and the variable stored is of logical type (TRUE, FALSE). They should be in capital letters. Anything that is typed after # is not run by R. They will be considered as a comment.

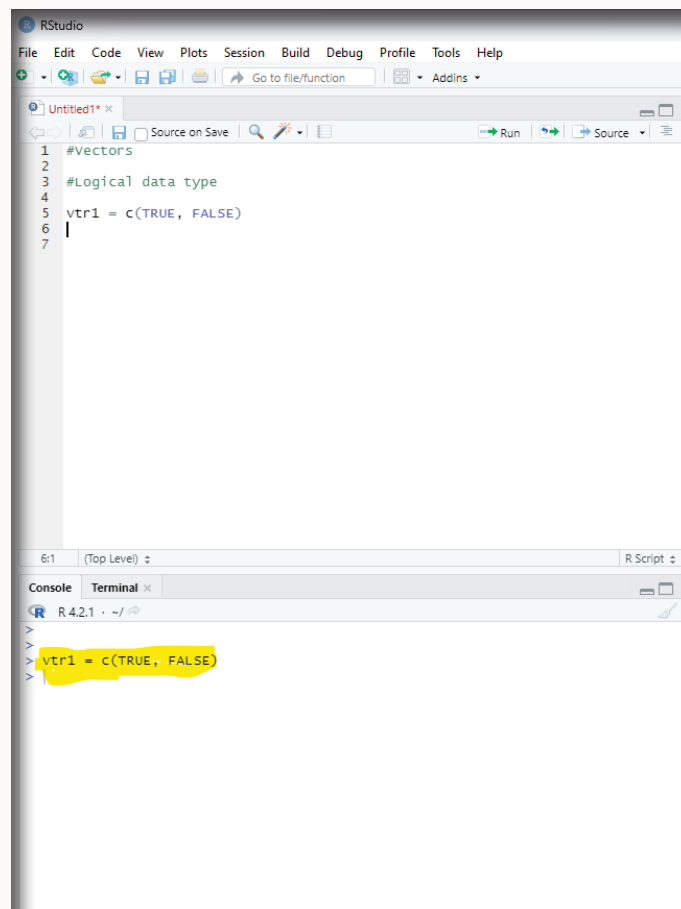
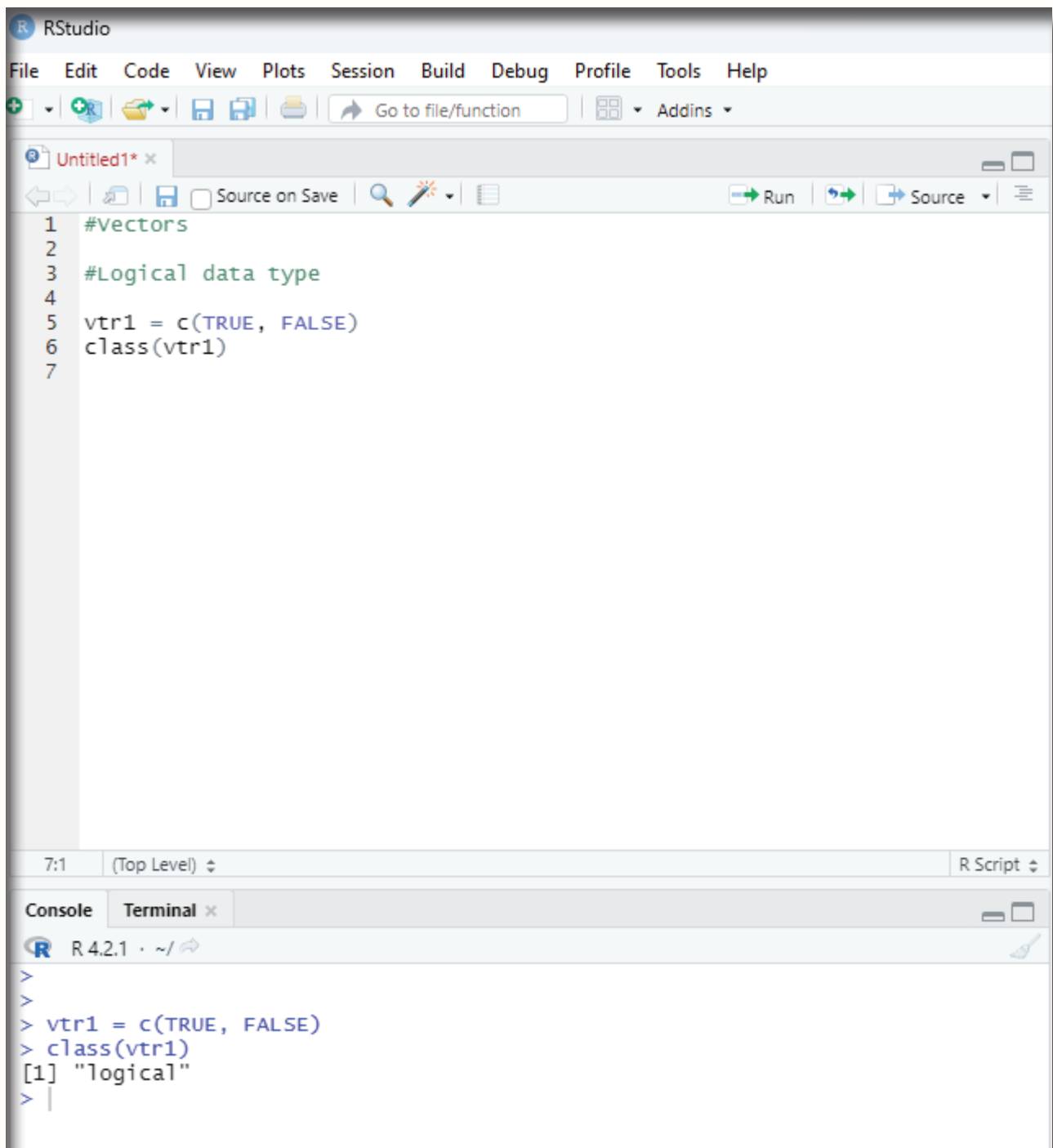


Image showing the result of clicking the run button. The result of running the code is displayed in the console window (highlighted yellow).

In order to ascertain what type of data has been allocated to the variable the class command would help.

Syntax for ascertaining the data type associated with a variable is: `class(name of the variable)`



The image shows the RStudio interface. The top menu bar includes File, Edit, Code, View, Plots, Session, Build, Debug, Profile, Tools, and Help. Below the menu is a toolbar with icons for file operations and a search bar labeled 'Go to file/function'. The main editor window, titled 'Untitled1* x', contains the following R code:

```
1 #vectors
2
3 #Logical data type
4
5 vtr1 = c(TRUE, FALSE)
6 class(vtr1)
7
```

Below the editor is a status bar showing '7:1 (Top Level)' and 'R Script'. At the bottom is a console window with tabs for 'Console' and 'Terminal'. The console shows the execution of the code:

```
>
>
> vtr1 = c(TRUE, FALSE)
> class(vtr1)
[1] "logical"
> |
```

Image showing command to ascertain the category of variable inside vector named vtr1. Note the output of the code run in the console window.

class(vtr1)

Output: [1] "logical"

Another vector is created. Name assigned to the newly created vector is vtr2.

```
vtr2 = c(15, 64.8777, 8888844)
```

In the newly created vector named vtr2 has the following data allocated to it;

```
15  
64.8777  
8888844
```

On pressing the run button in the scripting window the script is run and the output is displayed in the console window.

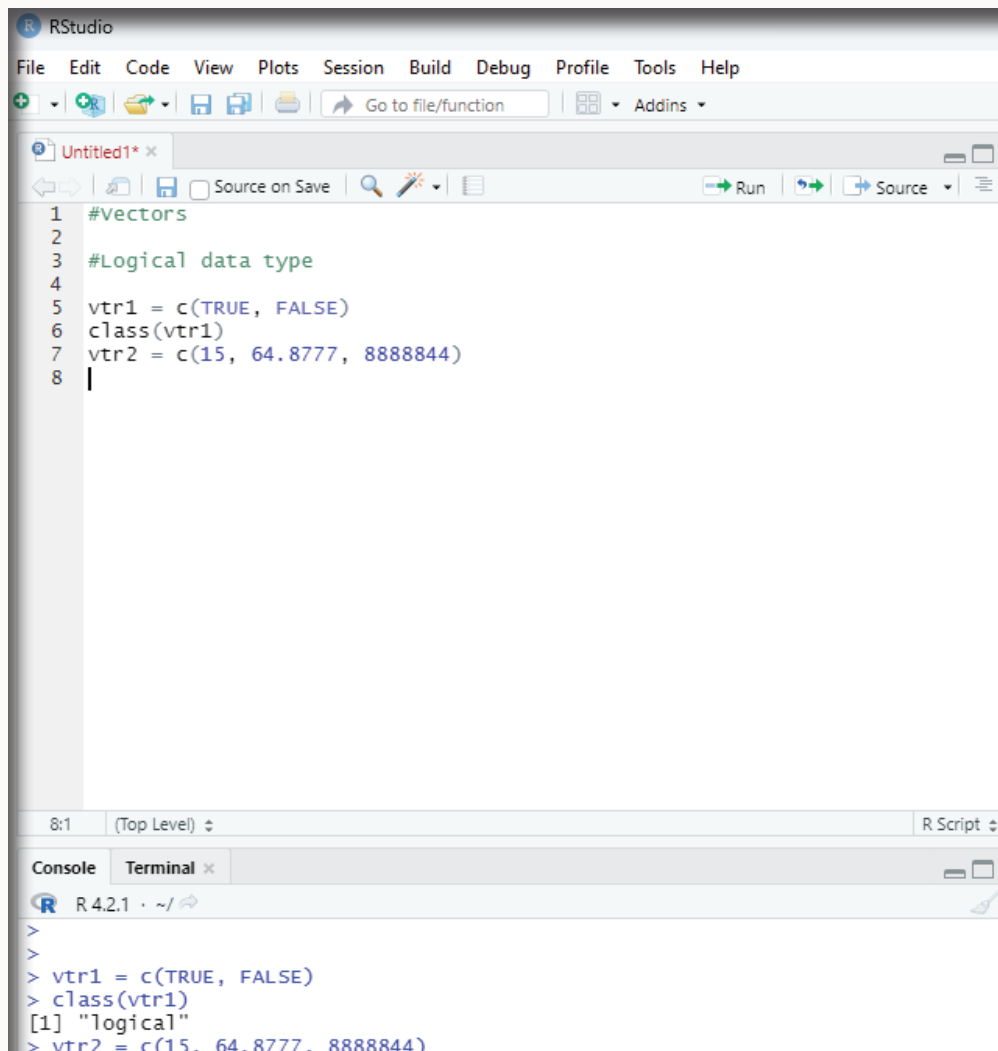


Image showing the second vector created and values allotted. Result is displayed in Console window

Values assigned to a vector can be seen by just keying the name of the vector in the console window and pressing the Enter key. For example the value stored in the vtr2 can be ascertained by keying in vtr2 in console window and pressing the Enter key.



```
R 4.2.1 · ~/
>
> vtr1 = c(TRUE, FALSE)
> class(vtr1)
[1] "logical"
> vtr2 = c(15, 64.8777, 8888844)
> vtr2
[1]      15.0000      64.8777 8888844.0000
>
```

Image showing the console window where a command to display value stored in vector 2 (vtr2) is displayed. Note the value is different from that of what was keyed in the scripting window. This is because one assigned value has four decimals and hence all the whole numbers are converted into decimals by adding four zeros after the whole number.

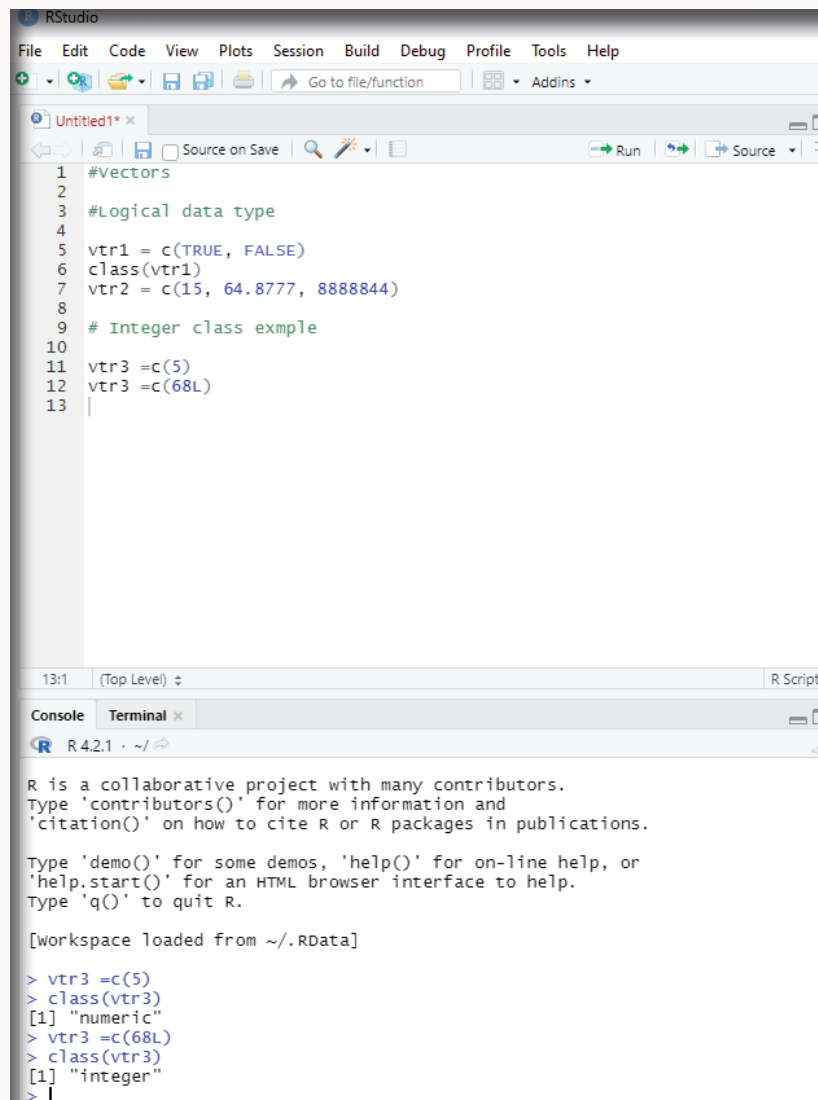
When the code for identifying the data type of vtr2 is keyed in the output displays Numeric.



```
R 4.2.1 · ~/
>
> vtr1 = c(TRUE, FALSE)
> class(vtr1)
[1] "logical"
> vtr2 = c(15, 64.8777, 8888844)
> vtr2
[1]      15.0000      64.8777 8888844.0000
> class(vtr2)
[1] "numeric"
>
```

Image showing the result of command class(vtr2). Type of data stored in the variable is displayed as numeric

Example for integer type data stored in a vector. If the user desires to store a value of 5 in a vector titled vtr3 then the class statement will describe the data as Numeric. If the stored number (whole number) is suffixed with 'L' then the class statement describes the data as an integer.



```
1 #vectors
2
3 #Logical data type
4
5 vtr1 = c(TRUE, FALSE)
6 class(vtr1)
7 vtr2 = c(15, 64.8777, 8888844)
8
9 # Integer class exmple
10
11 vtr3 =c(5)
12 vtr3 =c(68L)
13 |
```

```
R 4.2.1 · ~/
R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

[workspace loaded from ~/.RData]
> vtr3 =c(5)
> class(vtr3)
[1] "numeric"
> vtr3 =c(68L)
> class(vtr3)
[1] "integer"
> |
```

Image showing Integer class stored in a vector. Note only when the stored whole number is suffixed with a "L" the value will be recognized by R as an Integer. Note the console command class(vtr3) and its result.

```
vtr3 c=(5)
```

#This code is entered into the scripting window and run button is clicked.

The console window shows the result as being the value 5 assigned to the variable titled vtr3.

In the console window the following script is entered and run:

```
class(vtr3)
```

When the above command is keyed into the console window and run the result is displayed as:

```
[1] "numeric"
```

If the whole number entered into a variable is suffixed with "L" then it is considered as an Integer by R.

```
vtr3 c=(68L)
```

The above code is entered into the scripting window and run. This displays the result that the number 68L has been stored in the variable named vtr3.

```
class(vtr3)
```

The above command is given in the console window and run. This displays a class value as "Integer".

What will happen to the class if three types of variables included in a vector?

Code:

```
vtr5 = c(TRUE,35L,3.14)
```

vtr5 contains three types of variables:

1. Logical data
2. Integer
3. Numeric

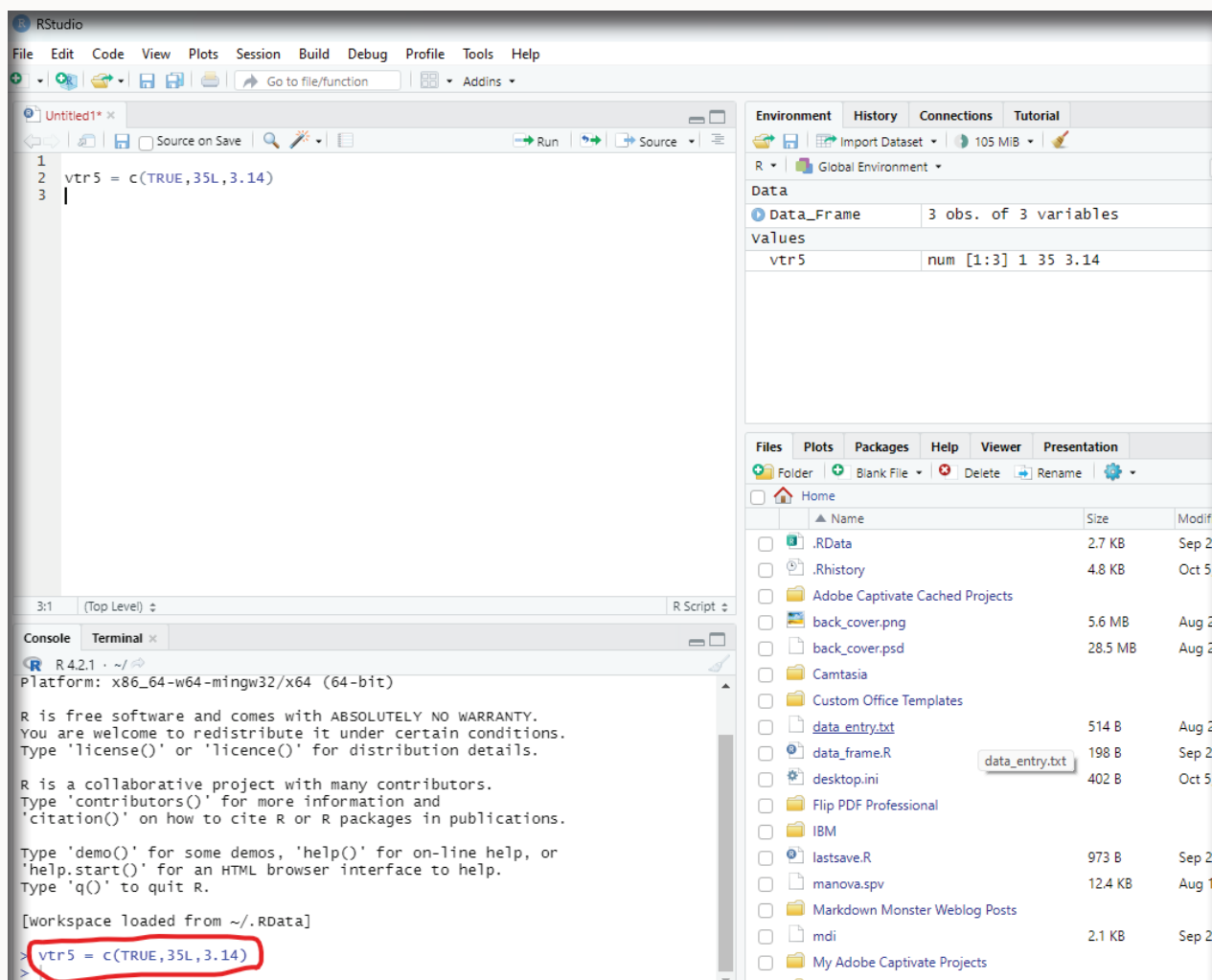
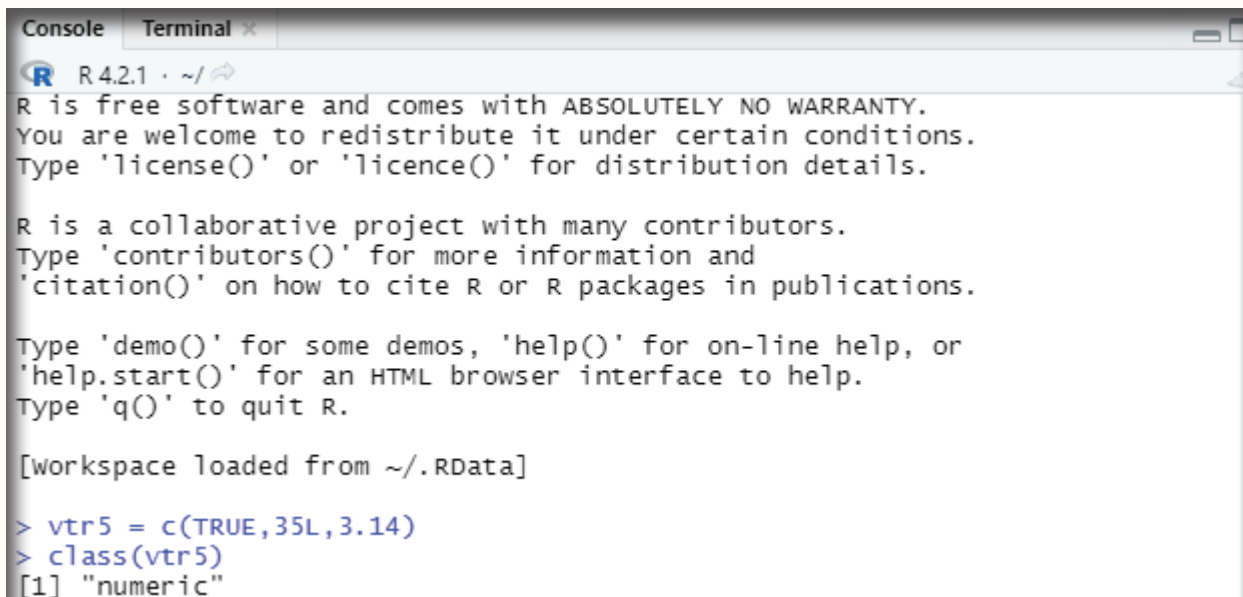


Image showing three types of data entered into a vector. The first one is the logical vector, the next one is an integer and the third one is the numeric. This code is entered into the scripting window. On clicking the run button the console window will demonstrate all these three data.

In the console window the following code can be used to ascertain the class of data:

class(vtr5)

On clicking the enter button the console displays as numeric the type of data.



```
R 4.2.1 ~/  
R is free software and comes with ABSOLUTELY NO WARRANTY.  
You are welcome to redistribute it under certain conditions.  
Type 'license()' or 'licence()' for distribution details.  
  
R is a collaborative project with many contributors.  
Type 'contributors()' for more information and  
'citation()' on how to cite R or R packages in publications.  
  
Type 'demo()' for some demos, 'help()' for on-line help, or  
'help.start()' for an HTML browser interface to help.  
Type 'q()' to quit R.  
  
[workspace loaded from ~/.RData]  
  
> vtr5 = c(TRUE,35L,3.14)  
> class(vtr5)  
[1] "numeric"
```

Image showing the console window when class query is used. It displays numeric as the type of data.

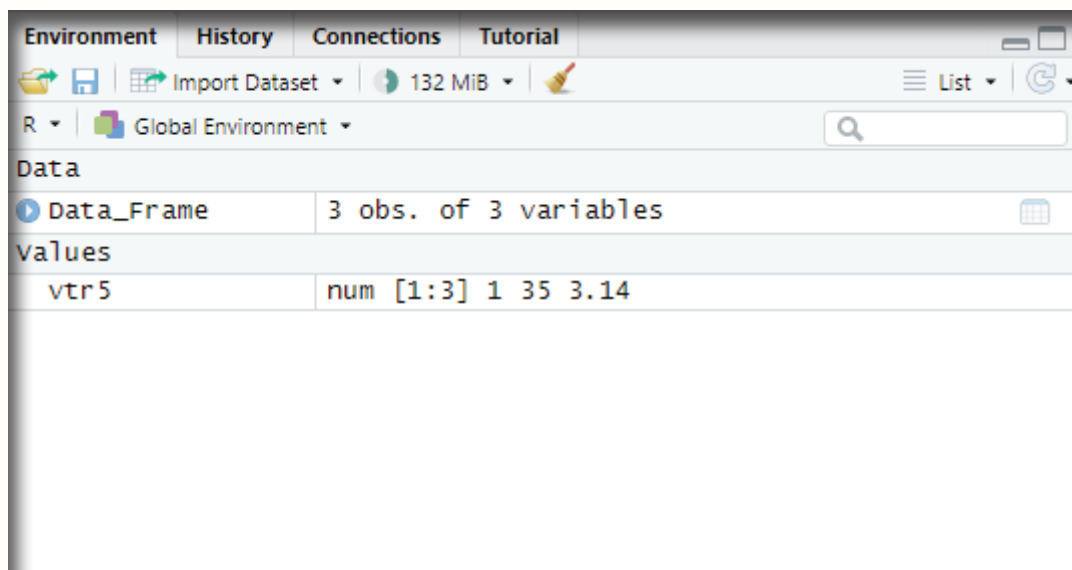
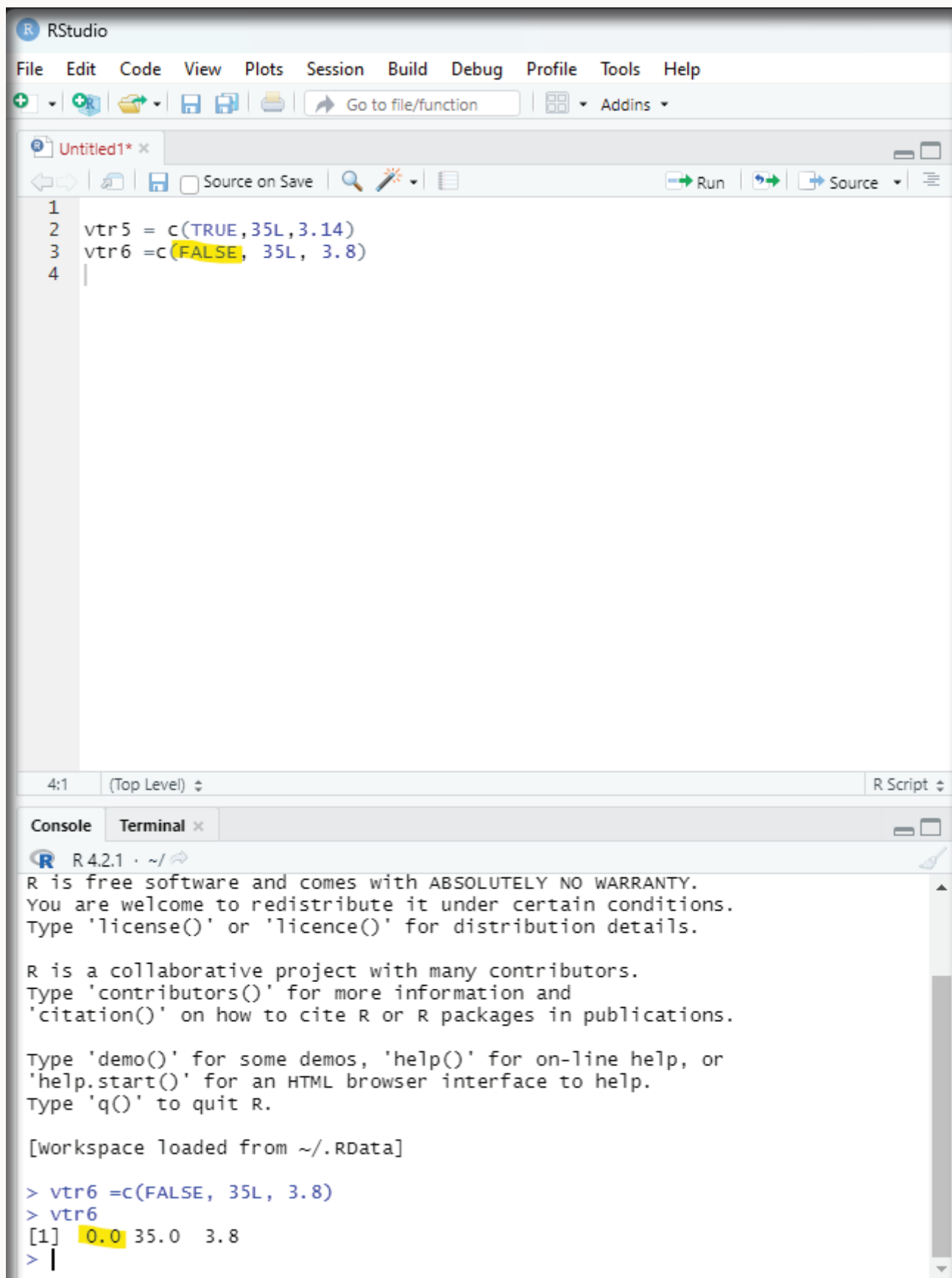


Image showing the Environment window where the name of the variable vtr5 is displayed and the class is displayed as numeric (num). It also says that there are three data (1:3) in the variable. Note 1 is used to instead of TRUE. Logical value TRUE has been assigned the numeric value of 1. When multiple types of data is entered into a vector, R software converts them into a unified data.

Now let us see what value would R assign if FALSE is used instead of TRUE:

The logical value of FALSE is assigned a value of 0 by R as shown in the image below.



The screenshot shows the RStudio interface. The source editor at the top contains the following R code:

```
1  
2 vtr5 = c(TRUE, 35L, 3.14)  
3 vtr6 = c(FALSE, 35L, 3.8)  
4 |
```

The console at the bottom shows the output of the code execution:

```
R 4.2.1 ~/  
R is free software and comes with ABSOLUTELY NO WARRANTY.  
You are welcome to redistribute it under certain conditions.  
Type 'license()' or 'licence()' for distribution details.  
  
R is a collaborative project with many contributors.  
Type 'contributors()' for more information and  
'citation()' on how to cite R or R packages in publications.  
  
Type 'demo()' for some demos, 'help()' for on-line help, or  
'help.start()' for an HTML browser interface to help.  
Type 'q()' to quit R.  
  
[workspace loaded from ~/.RData]  
  
> vtr6 = c(FALSE, 35L, 3.8)  
> vtr6  
[1] 0.0 35.0 3.8  
> |
```



```
vtr5 = c(TRUE, 35L, 3.14)
```

In this code the first value is Logical, the second is an integer and the third one is numeric. After these values have been assigned to vtr5 then when the class of the vector is queried for using the code `class(vtr5)`. The output generated would be “numeric”. This occurs because R converts all values into numeric data. The Logical data is also converted into numeric data, TRUE is assigned a value of 1. If it would have been FALSE then value 0 would be assigned.

```
vtr6 = c("hello", FALSE, 50L)
```

In this example code one can see that vr6 variable contains a character, Logical value and an Integer. On entering this data into the vector these values are created. Console window would reveal that all the data created are within double quotes. R considers all these values to be of character type. In other words it converts both logical and integer values to be character type. When different data types are entered into a variable then R converts them into a single data type.

Matrix:

This is quite similar to arrays in other programs. These are R objects in which the elements are arranged in a two dimensional rectangular layout.

Syntax for creating matrix in R:

```
matrix(data,nrow,ncol,byrow,dimnames)
```

- * data - it is the input vector which becomes the data elements of the matrix.
- * nrow - it is the number of rows to be created.
- * ncol - is the number of columns to be created.
- * byrow - is a logical clue. If true then the input vector elements are arranged by row.
- * dimname - is the names assigned to rows and columns.

code:

```
mtr = matrix(c(5:29),5,5,)
```

Using the above code a matrix is created with numbers between 5 and 29 with an increment of 1 between them. Number of Rows are specified as 5 and number of columns is specified as 5. This code is entered into the script window of RStudio. On clicking the Run button the values for the matrix get assigned successfully as seen from the output in the console window. On typing mtr (name of the matrix) in the console window and Enter button is clicked. Output demonstrates the arrangement of numbers between 5 and 29 in the form of matrix as shown in the figure.

The screenshot shows the RStudio interface. The source editor at the top contains the following R code:

```
1  
2  
3 mtr = matrix(c(5:29),5,5,)  
4 |  
5  
6  
7
```

The console at the bottom shows the execution of the code and the resulting matrix:

```
> mtr = matrix(c(5:29),5,5,)  
> mtr  
      [,1] [,2] [,3] [,4] [,5]  
[1,]    5   10   15   20   25  
[2,]    6   11   16   21   26  
[3,]    7   12   17   22   27  
[4,]    8   13   18   23   28  
[5,]    9   14   19   24   29  
> |
```

Image showing screen shot of the matrix code which is used to arrange numbers between 5 and 29 in 5 rows and 5 columns matrix

By default matrices are in column-wise order.

Another code for creating a matrix:

```
A = matrix(  
  
# Taking sequence of elements  
c(1, 2, 3, 4, 5, 6, 7, 8, 9),  
  
# No of rows  
nrow = 3,  
  
# No of columns  
ncol = 3,  
  
# By default matrices are in column-wise order  
# So this parameter decides how to arrange the matrix  
byrow = TRUE  
)  
  
# Naming rows  
rownames(A) = c("a", "b", "c")  
  
# Naming columns  
colnames(A) = c("c", "d", "e")  
  
cat("The 3x3 matrix:\n")  
print(A)
```

Creating a matrix where all rows and columns are filled by a single constant "k".

Note: Use of print command need not be used. It is sufficient to key in the variable name in the console window and pressing the Enter button will display the result. Use of print command or just the name of the variable is a personal choice of the programmer. Print syntax is introduced just to alert the reader that there are more than one way to instruct R to perform a task.

Syntax used is:

matrix(k,m,n)

k-the constant

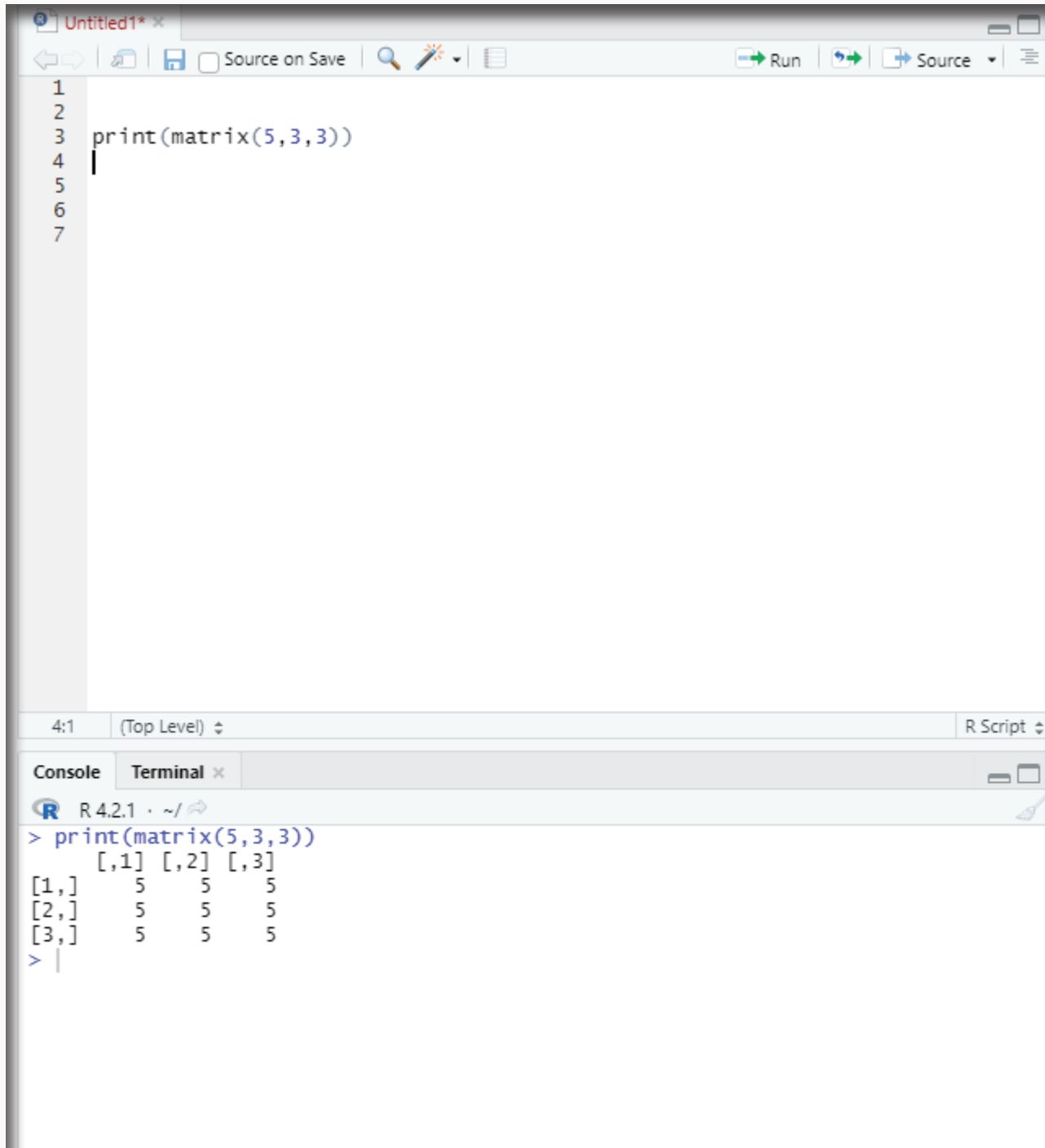
m-number of rows

n-number of columns

Code:

```
print(matrix(5,3,3))
```

On running this code R creates a 3x3 matrix with all values filled as 5.



```
1  
2  
3 print(matrix(5,3,3))  
4  
5  
6  
7
```

4:1 (Top Level) R Script

Console Terminal

```
R 4.2.1 ~/  
> print(matrix(5,3,3))  
      [,1] [,2] [,3]  
[1,]    5    5    5  
[2,]    5    5    5  
[3,]    5    5    5  
> |
```

Image showing matrix filled with the same number i.e., 5

Diagonal Matrix:

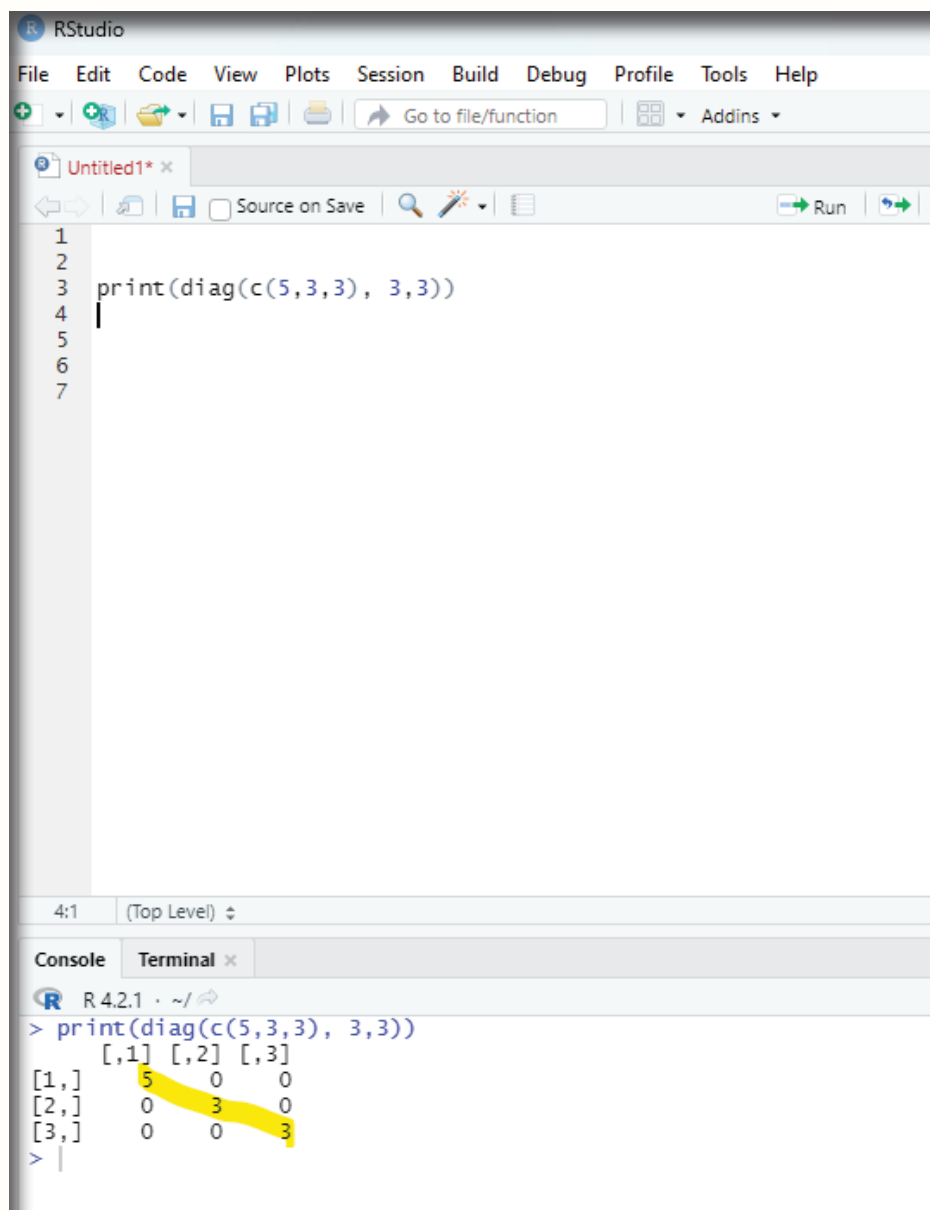
A diagonal matrix is a matrix in which the entries outside the main diagonal are 5,3,3.

Code:

#This diagonal matrix should have 3 rows and 3 columns.

Filled by array of elements (5,3,3).

print(diag(c(5,3,3), 3,3))



The screenshot shows the RStudio interface. The script editor contains the following code:

```
1  
2  
3 print(diag(c(5,3,3), 3,3))  
4  
5  
6  
7
```

The console output shows the result of the command:

```
> print(diag(c(5,3,3), 3,3))  
      [,1] [,2] [,3]  
[1,]    5    0    0  
[2,]    0    3    0  
[3,]    0    0    3
```

The diagonal elements (5, 3, 3) are highlighted with a yellow curved line.

Image showing a diagonal matrix with numbers 5,3, and 3 in the main diagonal

Identity matrix:

A square matrix in which all the elements of the principal diagonal are ones and all other elements are zeros. To create such a matrix the following syntax should be used:

Syntax:

`diag(k,m,n)`

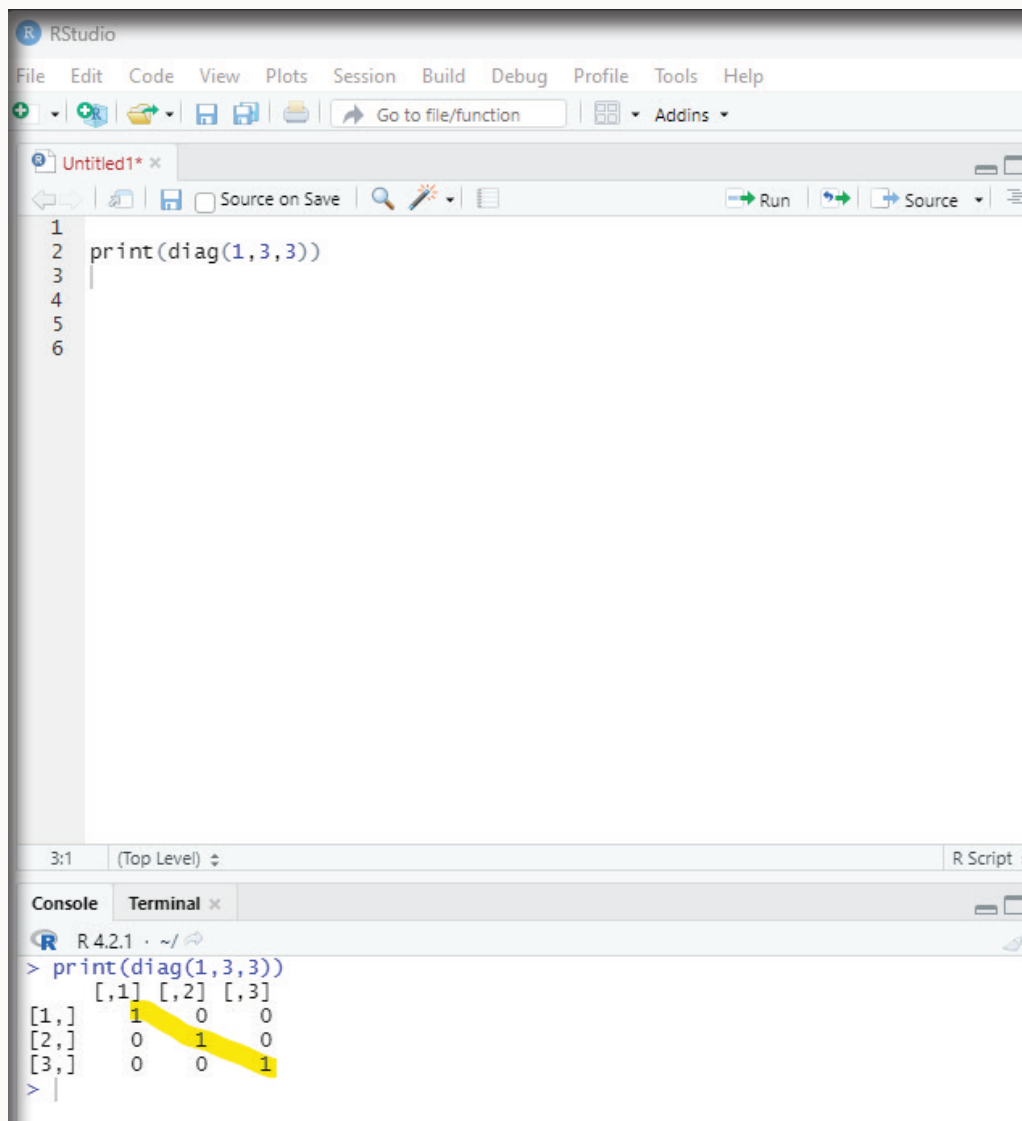
Parameters:

`k:1`

`m=no of rows`

`n=no of columns`

`print(diag(1,3,3))`



```
1  
2 print(diag(1,3,3))  
3  
4  
5  
6
```

```
> print(diag(1,3,3))  
[1,] 1 0 0  
[2,] 0 1 0  
[3,] 0 0 1
```

Image showing the result of code with 1 in the major diagonal and zero in all

Example for a matrix with 2 rows and three columns:

```
> A = matrix(  
+ c(2,4,3,1,5,7), # the data elements  
+ nrow=2,         # number of rows  
+ ncol=3,         # number of columns  
+ byrow = TRUE)  # fill matrix by rows  
  
> A              # Print the matrix
```

These examples help the reader to understand that there are various coding methodologies available in R Programming and it is for the programmer to choose which is best suited for them.

Assessing various elements in a matrix:

An element at the mth row, nth column of matrix A can be assessed by the expression A[m,n].

```
> A[2,3] # element at the 2nd row, 3rd column.
```

The entire mth row A can be extracted as A[m,].

```
> A [2] # 2nd row.
```

The entire nth column A can be extracted as A[,n].

```
> A[,3] # 3rd column
```

One can also extract more than one rows or columns at a time.

Matrix Construction:

There are various ways to construct a matrix. When one constructs a matrix directly with data elements, the matrix content is filled along the column orientation by default.

Example:

```
> B=matrix(  
+ c(2,4,3,1,5,7),  
+ nrow=3,  
+ ncol=2)
```

B has three rows and two columns

Transpose:

One can transpose a matrix by interchanging its column and rows with the function t.

```
>t(B) # transpose of B
```

Combining Matrices:

Columns of two matrices having the same number of rows can be combined into a larger matrix.

```
> c=matrix(  
+ c(7,4,2),  
+ nrow=3,  
+ ncol=1,
```

```
>c #c has 3 rows.
```

One can combine matrices B and C using cbind command.

```
> cbind(B, C)
```

One can also combine the rows of two matrices if they have the same number of columns with rbind function.

```
> D=matrix(  
+ c(6,2),  
+ nrow=1,  
+ ncol=2)
```

```
>D # D has 2 columns
```

```
>rbind(B,D)
```

Deconstruction:

The user can deconstruct a matrix by applying c function which combines all the column vectors into one.

```
>c(B)
```


Arrays:

These are R data objects which can store data in more than two dimensions. Only precondition being that the different data should be of the same class.

Syntax used:

`array(data,dim,dimnames)`

`array(c(0:15), dim=c(4,4,2,2))`

Basically 64 elements are stored in 4 different matrices.

If the number of values is less than the number of arrays / matrix then it takes the same input vector and starts to insert elements already inserted.

```
1 array(c(0:15), dim=c(4,4,2,2))
2
3
4
5
6
```

```
> array(c(0:15), dim=c(4,4,2,2))
, , 1, 1
      [,1] [,2] [,3] [,4]
[1,]    0    4    8   12
[2,]    1    5    9   13
[3,]    2    6   10   14
[4,]    3    7   11   15
, , 2, 1
      [,1] [,2] [,3] [,4]
[1,]    0    4    8   12
[2,]    1    5    9   13
[3,]    2    6   10   14
[4,]    3    7   11   15
```

Image showing an array with numbers ranging from 0 to 15 created. It has 4 columns, four rows and 4 dimensions

```

>
> array(c(0:15), dim=c(4,4,2,2) )
, , 1, 1
      [,1] [,2] [,3] [,4]
[1,]  0   4   8  12
[2,]  1   5   9  13
[3,]  2   6  10  14
[4,]  3   7  11  15

, , 2, 1
      [,1] [,2] [,3] [,4]
[1,]  0   4   8  12
[2,]  1   5   9  13
[3,]  2   6  10  14
[4,]  3   7  11  15

, , 1, 2
      [,1] [,2] [,3] [,4]
[1,]  0   4   8  12
[2,]  1   5   9  13
[3,]  2   6  10  14
[4,]  3   7  11  15

, , 2, 2
      [,1] [,2] [,3] [,4]
[1,]  0   4   8  12
[2,]  1   5   9  13
[3,]  2   6  10  14
[4,]  3   7  11  15

```

Seen above are the four columns and rows arranged in four dimensions.

Two vectors containing similar objects can be combined into one array.

Example:

```
# Creating two vectors of different lengths.
```

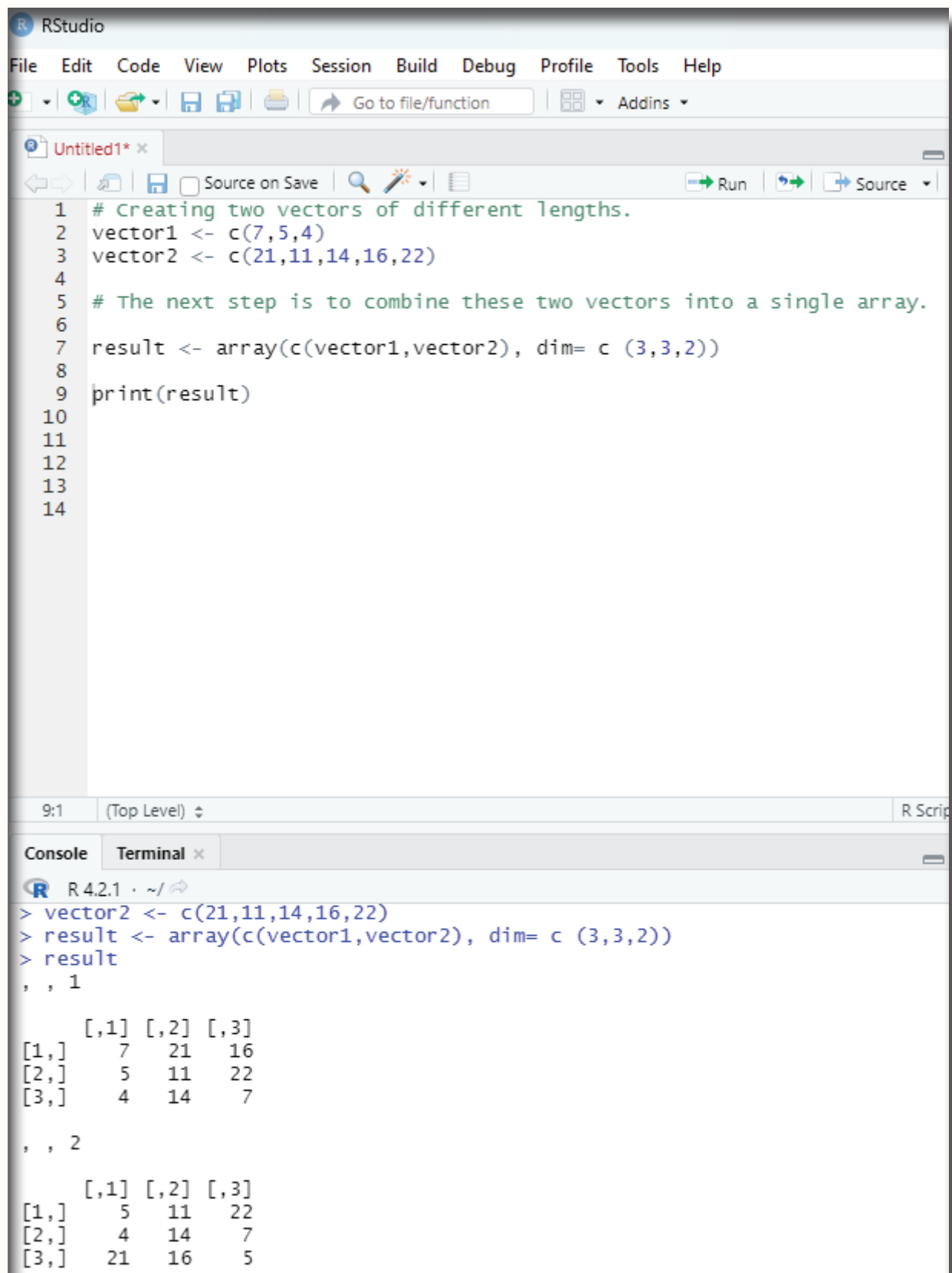
```
vector1 <- c(7,5,4)
```

```
vector2 <- c(21,11,14,16,22)
```

```
# The next step is to combine these two vectors into a single array.
```

```
result <- array(c(vector1,vector2), dim= c (3,3,2))
```

```
print(result)
```



The screenshot shows the RStudio interface. The source editor contains the following R code:

```
1 # Creating two vectors of different lengths.
2 vector1 <- c(7,5,4)
3 vector2 <- c(21,11,14,16,22)
4
5 # The next step is to combine these two vectors into a single array.
6
7 result <- array(c(vector1,vector2), dim= c (3,3,2))
8
9 print(result)
```

The console output shows the execution of the code:

```
> vector2 <- c(21,11,14,16,22)
> result <- array(c(vector1,vector2), dim= c (3,3,2))
> result
, , 1
  [,1] [,2] [,3]
[1,]   7  21  16
[2,]   5  11  22
[3,]   4  14   7

, , 2
  [,1] [,2] [,3]
[1,]   5  11  22
[2,]   4  14   7
[3,]  21  16   5
```

Image showing two vectors with data of different sizes combined into a single array with two dimensions

Columns and Rows in array can be named using dimnames parameter.

Example:

Step 1 - Create two vectors of different lengths.

```
vector1 <-c(3,4,8)
```

```
vector2 <-c(10,13,11,22,34,22)
```

```
column.names <-c("COL1", "COL2", "COL3")
```

```
row.names <-c("ROW1", "ROW2", "ROW3")
```

```
matrix.names <-c("matrix1", "matrix2")
```

Step 2 - Combine these vectors as input into the array.

```
result <-array(c(vector1,vector2),dim = c(3,3,2),dimnames=list(row.names,column.names,ma-  
trix.names)
```

```
print(result)
```

Note the command list is used here. It will be discussed later in the chapter.

One can assess the elements in the array using the following command:

To print the third row of the second matrix of the array

```
print(result[3,,2])
```

To print the element in the 1st row and 3rd column of the 1st matrix.

```
print(result[1,3,1])
```

In order to print the entire two matrices.

```
print(result[,,2])
```

Manipulating elements within array:

Since array is made up of matrices in multiple dimensions, the operations on elements of array can be carried out by accessing elements of the matrices.

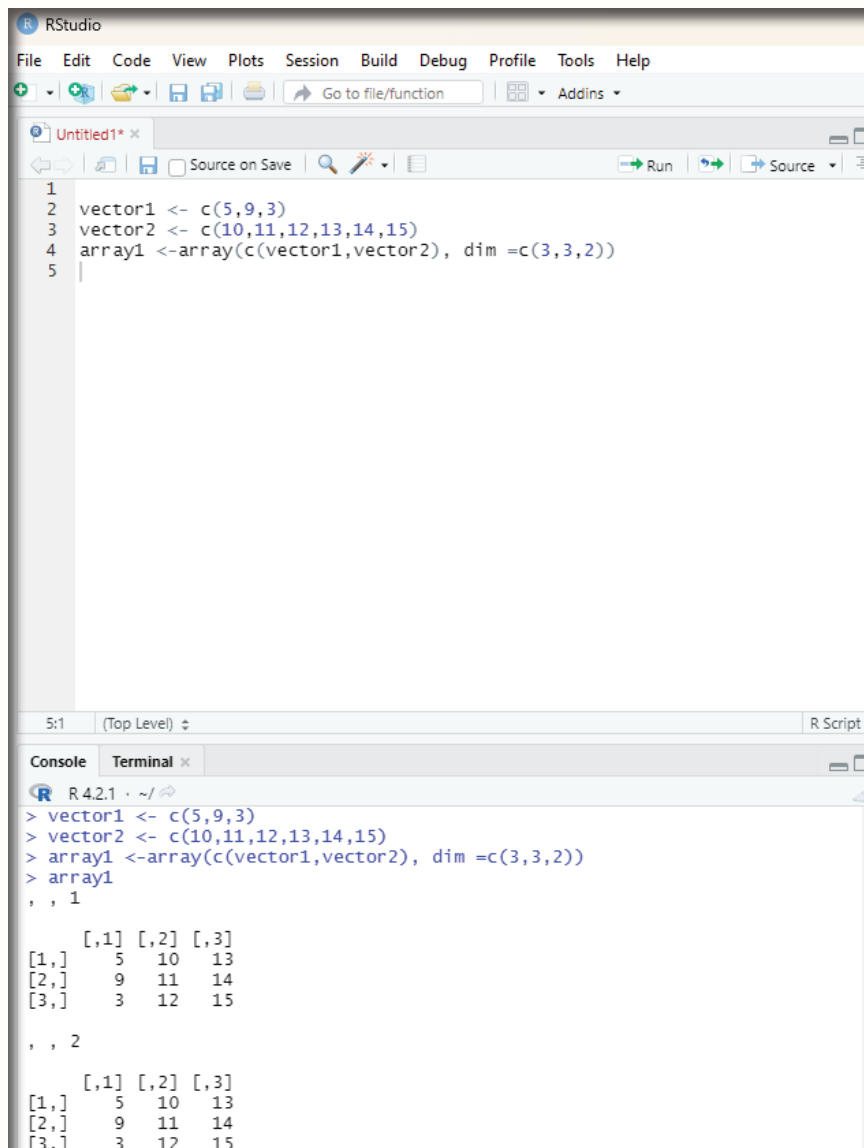
Create two vectors of different lengths.

vector1 <- c(5,9,3)

vector2 <- c(10,11,12,13,14,15)

Take these vectors as input into the array.

array1 <-array(c(vector1,vector2), dim =c(3,3,2))



The screenshot shows the RStudio interface. The script editor contains the following code:

```
1  
2 vector1 <- c(5,9,3)  
3 vector2 <- c(10,11,12,13,14,15)  
4 array1 <-array(c(vector1,vector2), dim =c(3,3,2))  
5
```

The console output shows the execution of the code:

```
> vector1 <- c(5,9,3)  
> vector2 <- c(10,11,12,13,14,15)  
> array1 <-array(c(vector1,vector2), dim =c(3,3,2))  
> array1  
, , 1  
  
  [,1] [,2] [,3]  
[1,]  5  10  13  
[2,]  9  11  14  
[3,]  3  12  15  
  
, , 2  
  
  [,1] [,2] [,3]  
[1,]  5  10  13  
[2,]  9  11  14  
[3,]  3  12  15
```

Image showing array1 being generated using the code specified

Create two vectors of different lengths.

vector3 <- c(9,1,0)

vector4 <- c(6,0,11,3,14,1,2,6,9)

array2 <- array (c(vector3,vector4),dim =c(3,3,2))

Create matrices from these arrays.

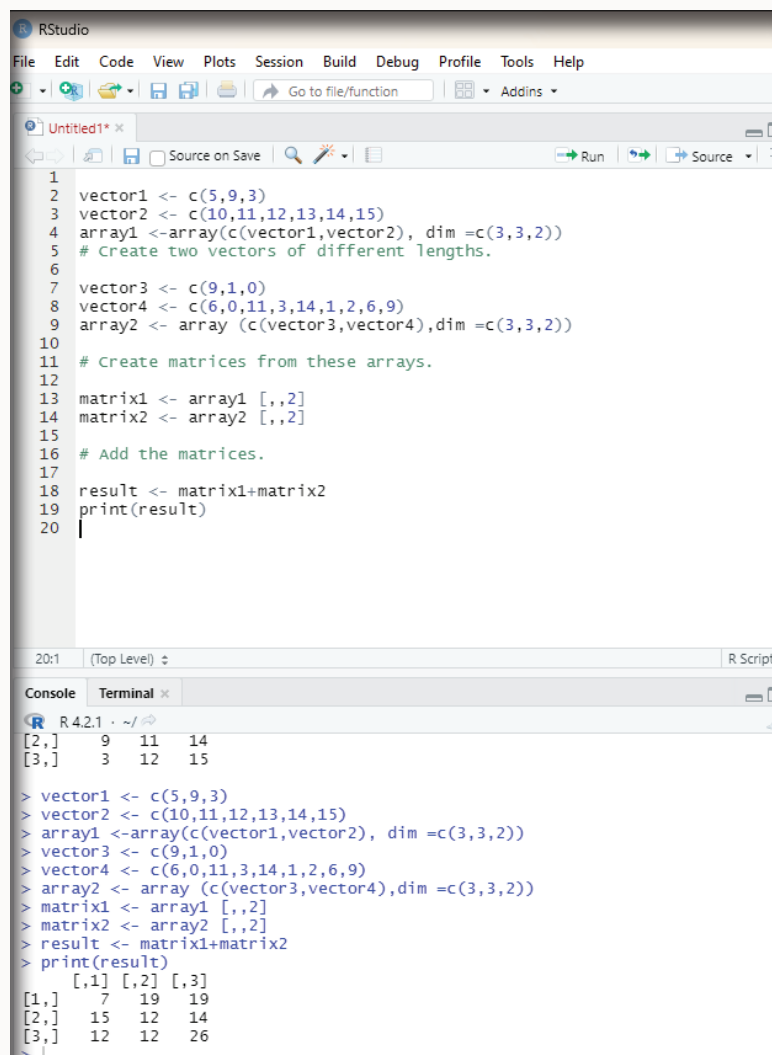
matrix1 <- array1 [,2]

matrix2 <- array2 [,2]

Add the matrices.

result <- matrix1+matrix2

print(result)



The screenshot shows the RStudio interface. The script editor contains the following code:

```
1  
2 vector1 <- c(5,9,3)  
3 vector2 <- c(10,11,12,13,14,15)  
4 array1 <-array(c(vector1,vector2), dim =c(3,3,2))  
5 # Create two vectors of different lengths.  
6  
7 vector3 <- c(9,1,0)  
8 vector4 <- c(6,0,11,3,14,1,2,6,9)  
9 array2 <- array (c(vector3,vector4),dim =c(3,3,2))  
10  
11 # Create matrices from these arrays.  
12  
13 matrix1 <- array1 [,2]  
14 matrix2 <- array2 [,2]  
15  
16 # Add the matrices.  
17  
18 result <- matrix1+matrix2  
19 print(result)  
20
```

The console output shows the initial dimensions of the arrays and the final result of the matrix addition:

```
R 4.2.1 ~/  
[2,] 9 11 14  
[3,] 3 12 15  
  
> vector1 <- c(5,9,3)  
> vector2 <- c(10,11,12,13,14,15)  
> array1 <-array(c(vector1,vector2), dim =c(3,3,2))  
> vector3 <- c(9,1,0)  
> vector4 <- c(6,0,11,3,14,1,2,6,9)  
> array2 <- array (c(vector3,vector4),dim =c(3,3,2))  
> matrix1 <- array1 [,2]  
> matrix2 <- array2 [,2]  
> result <- matrix1+matrix2  
> print(result)  
[1,] [2,] [3,]  
[1,] 7 19 19  
[2,] 15 12 14  
[3,] 12 12 26  
>
```

Image showing the results of adding two matrices

Calculations can be performed across array elements:

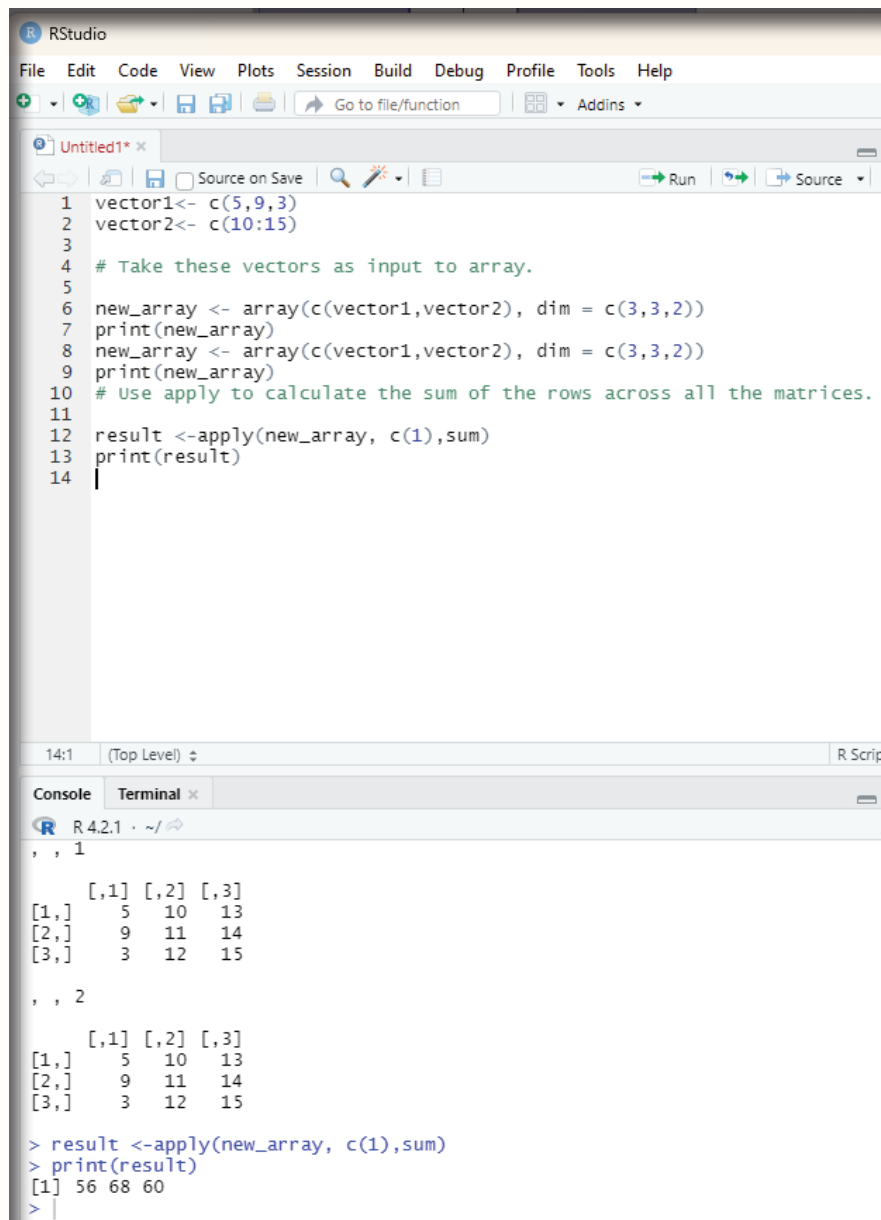
One can perform calculations across the elements in an array using the following syntax:

apply(x, margin, fun)

x is an array

margin is the name of the data set used

fun is the function to be applied across the elements in the array.



```
1 vector1<- c(5,9,3)
2 vector2<- c(10:15)
3
4 # Take these vectors as input to array.
5
6 new_array <- array(c(vector1,vector2), dim = c(3,3,2))
7 print(new_array)
8 new_array <- array(c(vector1,vector2), dim = c(3,3,2))
9 print(new_array)
10 # Use apply to calculate the sum of the rows across all the matrices.
11
12 result <-apply(new_array, c(1),sum)
13 print(result)
14 |
```

The console output shows the array structure and the result of the apply() function:

```
, , 1
      [,1] [,2] [,3]
[1,]    5   10   13
[2,]    9   11   14
[3,]    3   12   15

, , 2
      [,1] [,2] [,3]
[1,]    5   10   13
[2,]    9   11   14
[3,]    3   12   15

> result <-apply(new_array, c(1),sum)
> print(result)
[1] 56 68 60
> |
```

Image showing apply() command used to perform calculations

```
# Create two vectors of different lengths.
```

```
vector1<- c(5,9,3)
```

```
vector2<- c(10:15)
```

```
# Take these vectors as input to array.
```

```
new_array <- array(c(vector1,vector2), dim = c(3,3,2))
```

```
print(new_array)
```

```
# Use apply to calculate the sum of the rows across all the matrices.
```

```
result <-apply(new_array, c(1),sum)
```

```
print(result)
```

Displayed result is shown below:

```
, 1
     [,1] [,2] [,3]
[1,]   5  10  13
[2,]   9  11  14
[3,]   3  12  15

, 2
     [,1] [,2] [,3]
[1,]   5  10  13
[2,]   9  11  14
[3,]   3  12  15

[1] 56 68 60
```

R Factors:

Factors are data objects that are used to categorize the data and store it as levels. They can store both strings and integers. They are useful in the columns which have a limited number of unique values (like male, female, true, false etc). They are useful in statistical analysis for statistical modeling.

Factors can be created using factor() function by taking vector as an input.


```
# Create a vector as input.
```

```
data <-c("East", "West", "East", "North", "East", "West", "West", "West", "East", "North")
```

```
print (data)
```

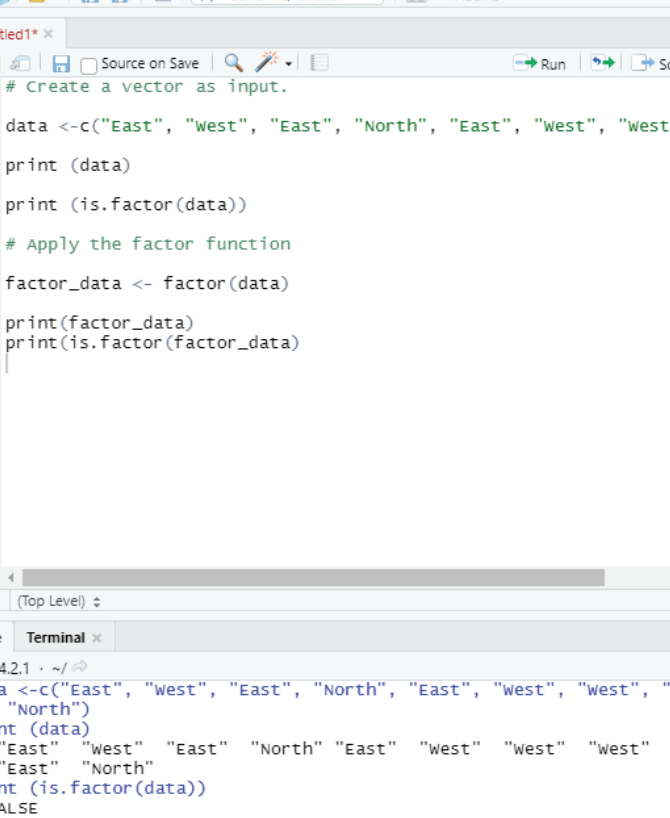
```
print(is.factor(data))
```

Apply the factor function

```
factor_data <- factor(data)
```

```
print(factor_data)
```

```
print(is.factor(factor_data))
```



The screenshot shows the RStudio interface. The top menu bar includes File, Edit, Code, View, Plots, Session, Build, Debug, Profile, Tools, and Help. Below the menu is a toolbar with icons for adding files, saving, and running code. The main editor window, titled 'Untitled1', contains the following R code:

```

1 # Create a vector as input.
2
3 data <-c("East", "West", "East", "North", "East", "West", "West", "West"
4
5 print (data)
6
7 print (is.factor(data))
8
9 # Apply the factor function
10
11 factor_data <- factor(data)
12
13 print(factor_data)
14 print(is.factor(factor_data))
15
16

```

At the bottom, the console window shows the output of the code:

```

> data <-c("East", "West", "East", "North", "East", "West", "West", "West", "East", "North")
> print (data)
[1] "East" "West" "East" "North" "East" "West" "West" "West"
[9] "East" "North"
> print (is.factor(data))
[1] FALSE
> factor_data <- factor(data)
> print(factor_data)
[1] East West East North East West West West East North
Levels: East North West
> print(is.factor(factor_data))
+

```

Image showing use of R factor

There are two steps involved in creating a factor:

1. Creating a vector
2. converting the created vector into a factor using the function factor()

The user desires to create a factor gender with two levels i.e., male and female.

```
# Creating a vector
```

```
x<-c("Female", "Male", "Male", "Female")
```

```
print (x)
```

```
# Converting the vector x into a factor.
```

```
# named gender
```

```
gender <-factor(x)
```

```
print(gender)
```

Output:

```
> x<-c("Female", "Male", "Male", "Female")
> print (x)
[1] "Female" "Male"  "Male"  "Female"
> gender <-factor(x)
> print(gender)
[1] Female Male  Male  Female
Levels: Female Male
```

One can use the function levels() to check the level of the factor.

Accessing elements of a Factor in R:

It is something like accessing elements of a vector. The same principle is used to access the elements of a factor.

```
gender <- factor (c( "female", "male", "male", "female", "female"));
```

```
gender [3]
```

Output:

```
[1] male
Levels: female male
```

The screenshot shows the RStudio interface. The top menu bar includes File, Edit, Code, View, Plots, Session, Build, Debug, Profile, Tools, and Help. Below the menu is a toolbar with icons for file operations and a search bar. The main editor window, titled 'Untitled1* x', contains the following R code:

```
1
2
3 # Creating a vector
4
5 x<-c("Female", "Male", "Male", "Female")
6
7 print (x)
8
9 # Converting the vector x into a factor.
10 # named gender
11
12 gender <-factor(x)
13 print(gender)
14
```

Below the editor is a status bar showing '14:1 (Top Level) R Script'. At the bottom is a console window with the following output:

```
R 4.2.1 ~/  
> x<-c("Female", "Male", "Male", "Female")  
> print (x)  
[1] "Female" "Male"  "Male"  "Female"  
> gender <-factor(x)  
> print(gender)  
[1] Female Male  Male  Female  
Levels: Female Male
```

Image showing another example of the use of R factor

More than one element can also be accessed at a time.

```
gender[c(2,4)]
```

Output:

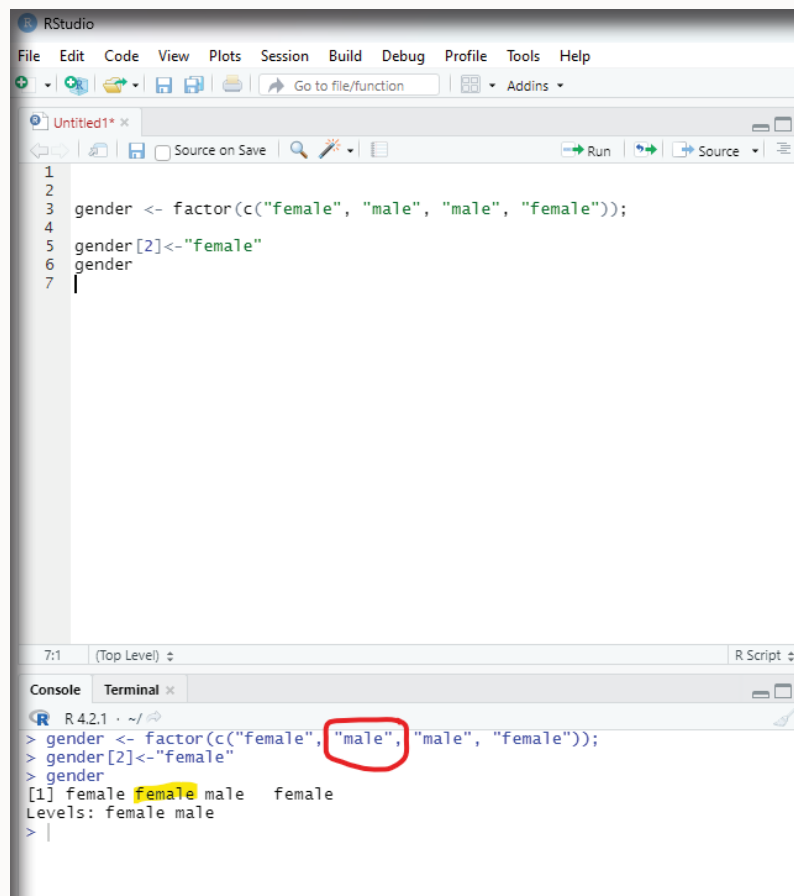
```
[1] male female  
Levels: female male
```

Modification of a factor in R:

After forming a factor, its components can be modified. The new values that needs to be assigned must be at the predefined level. If the value is gender then the new value should also be gender.

```
gender <- factor(c("female", "male", "male", "female"));
```

```
gender[2]<-"female"  
gender
```



The screenshot shows the RStudio interface. The script editor contains the following code:

```
1  
2  
3 gender <- factor(c("female", "male", "male", "female"));  
4  
5 gender[2]<-"female"  
6 gender  
7
```

The console output shows the result of the code execution:

```
> gender <- factor(c("female", "male", "male", "female"));  
> gender[2]<-"female"  
> gender  
[1] female female male female  
Levels: female male
```

In the console output, the word "female" in the second position of the factor vector is highlighted in yellow. In the script editor, the word "male" in the second position of the factor vector is circled in red.

Image showing modification of a factor in R

Output:

```
[1] female female male  female
```

```
Levels: female male
```

The user can also add a new level to the factor.

In this example a new level “other” needs to be added to the gender.

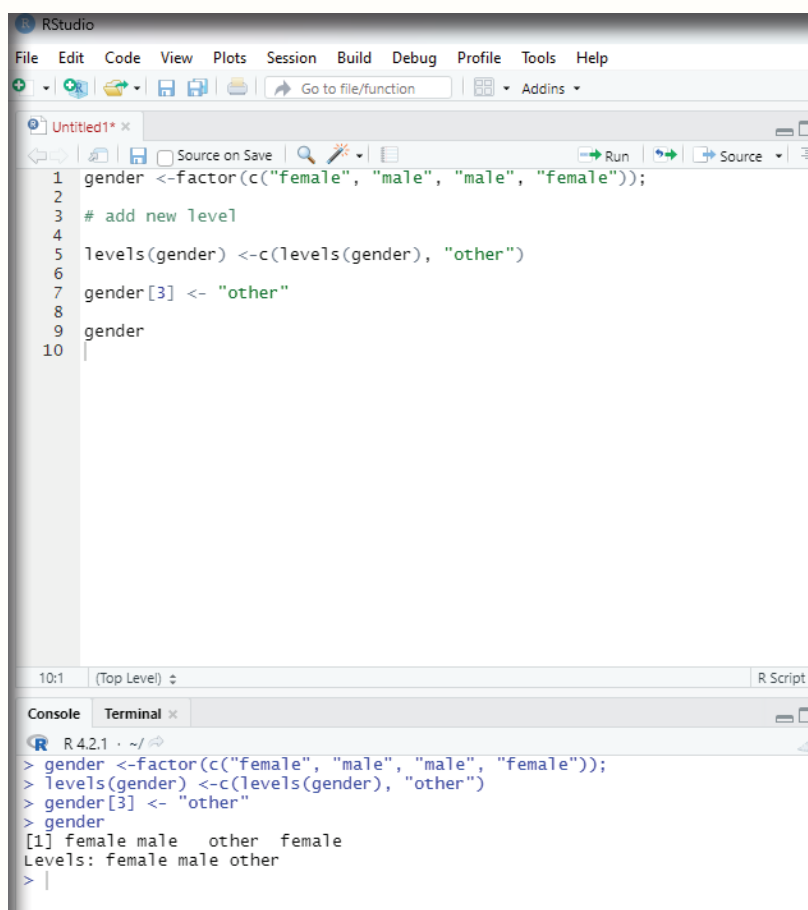
```
gender <-factor(c("female", "male", "male", "female"));
```

```
# add new level
```

```
levels(gender) <-c(levels(gender), "other")
```

```
gender[3] <- "other"
```

```
gender
```



```
RStudio
File Edit Code View Plots Session Build Debug Profile Tools Help
+ - Go to file/function
Untitled1* x
1 gender <-factor(c("female", "male", "male", "female"));
2
3 # add new level
4
5 levels(gender) <-c(levels(gender), "other")
6
7 gender[3] <- "other"
8
9 gender
10 |

10:1 (Top Level) R Script
Console Terminal x
R 4.2.1 ~
> gender <-factor(c("female", "male", "male", "female"));
> levels(gender) <-c(levels(gender), "other")
> gender[3] <- "other"
> gender
[1] female male  other female
Levels: female male other
> |
```

Image showing adding a new level to a factor

Output:

```
[1] female male  other female  
Levels: female male other
```

Lists:

These are the R objects that contain elements of different data types like - number, strings, vectors and another list inside it.

Syntax - `list(data)`

Example: Running this code in the script window will provide the list of the elements inside the three vectors in the list.

```
vtr1 <-c(1:5)
```

```
vtr2 <-c("hi", "hello", "How are you")
```

```
vtr3 <-c(TRUE,TRUE,FALSE,FALSE)
```

```
myList <-(vtr1,vtr2,vtr3)
```

Using List function all data retain their original data type. They don't get converted into common data format. If the user wants to use multiple data types without resorting to conversion to a common data type then list function should be used.

Syntax - `list(data)`

Example: Running this code in the script window will provide the list of the elements inside the three vectors in the list.

```
vtr1 <-c(1:5)
```

```
vtr2 <-c("hi", "hello", "How are you")
```

```
vtr3 <-c(TRUE,TRUE,FALSE,FALSE)
```

```
myList <-(vtr1,vtr2,vtr3)
```

Using List function all data retain their original data type. They don't get converted into common data format. If the user wants to use multiple data types without resorting to conversion to a common data type then list function should be used.

Syntax - list(data)

Example: Running this code in the script window will provide the list of the elements inside the three vectors in the list.

```
vtr1 = c(1:5)
```

```
vtr2 = c("hi", "hello", "How are you")
```

```
vtr3 = c(TRUE,TRUE,FALSE,FALSE)
```

```
myList <-c(vtr1,vtr2,vtr3)
```

Using List function all data retain their original data type. They don't get converted into common data format. If the user wants to use multiple data types without resorting to conversion to a common data type then list function should be used.

Note:

Various assignment operators are used in this example. They include =c and <-c. This is just to indicate both these operators can be used interchangeably. Operators will be discussed in detail in ensuing chapters.

Output:

```
myList
[1] "1"      "2"      "3"      "4"
[5] "5"      "hi"     "hello"  "How are you"
[9] "TRUE"   "TRUE"   "FALSE"  "FALSE"
```

Using List function all data retain their original data type. They don't get converted into common data format. If the user wants to use multiple data types without resorting to conversion to a common data type then list function should be used.

A list can also contain a matrix or a function as its elements. List is created using list() function.

Example:

Create a list containing strings, numbers, vectors and # a logical value.

```
list_data <- list("Red", "Green","Blue", c(21,32,11), TRUE, 51.23, 119.1)
print(list_data)
```

The image shows the RStudio interface. The top menu bar includes File, Edit, Code, View, Plots, Session, Build, Debug, Profile, Tools, and Help. Below the menu is a toolbar with icons for file operations and a search bar labeled 'Go to file/function'. The main editor window, titled 'Untitled1*', contains the following R code:

```
1 vtr1 = c(1:5)
2
3 vtr2 = c("hi", "hello", "How are you")
4
5 vtr3 = c(TRUE, TRUE, FALSE, FALSE)
6
7 myList = c(vtr1, vtr2, vtr3)
8
9
```

Below the editor is a console window showing the execution of the code. The prompt is 'R 4.2.1 · ~/'. The output of the code is displayed as follows:

```
> vtr1 = c(1:5)
> vtr2 = c("hi", "hello", "How are you")
> vtr3 = c(TRUE, TRUE, FALSE, FALSE)
> myList = c(vtr1, vtr2, vtr3)
>
> myList
[1] "1"      "2"      "3"      "4"
[5] "5"      "hi"     "hello"  "How are you"
[9] "TRUE"   "TRUE"   "FALSE"  "FALSE"
```

Image showing a list being formed with vectors containing numerical values, character values and Logical values.

Output:

```
[[1]]  
[1] "Red"  
  
[[2]]  
[1] "Green"  
  
[[3]]  
[1] "Blue"  
  
[[4]]  
[1] 21 32 11  
  
[[5]]  
[1] TRUE  
  
[[6]]  
[1] 51.23  
  
[[7]]  
[1] 119.1
```

Output:

```
$`1st Quarter`  
[1] "March" "April" "June"  
  
$A_Matrix  
  [,1] [,2] [,3]  
[1,]  4   3  10  
[2,]  6  -1   7  
  
$`A Inner list`  
$`A Inner list`[[1]]  
[1] "Yellow"  
  
$`A Inner list`[[2]]  
[1] 11.2
```

Naming List elements:

The list elements can be given names and they can be accessed using these names.

The image shows the RStudio interface. The top menu bar includes File, Edit, Code, View, Plots, Session, Build, Debug, Profile, Tools, and Help. Below the menu is a toolbar with icons for file operations and a search bar. The main editor window, titled 'Untitled1*', contains the following R code:

```
1  
2 # Create a list containing a vector, a matrix and a list.  
3  
4 list_data <- list(c("March", "April", "June"), matrix (c(4,6,3,-1,10,7),  
5  
6 # Provide names to the elements in the list.  
7  
8 names(list_data) <-c ("1st Quarter", "A_Matrix", "A Inner list")  
9  
10 # show the list.  
11  
12 print(list_data)  
13
```

The console window at the bottom shows the output of the code:

```
R 4.2.1 ~/  
> print(list_data)  
$`1st Quarter`  
[1] "March" "April" "June"  
  
$A_Matrix  
  [,1] [,2] [,3]  
[1,]   4   3  10  
[2,]   6  -1   7  
  
$`A Inner list`  
$`A Inner list`[[1]]  
[1] "Yellow"  
  
$`A Inner list`[[2]]  
[1] 11.2
```

Image showing a list containing a vector and a matrix

The image shows the RStudio interface. The top menu bar includes File, Edit, Code, View, Plots, Session, Build, Debug, Profile, Tools, and Help. Below the menu is a toolbar with icons for file operations and a search bar labeled 'Go to file/function'. The main editor window, titled 'Untitled1* x', contains the following R code:

```
1 # Create a list containing a vector, a matrix and a list.
2
3 list_data <- list(c("March", "April", "June"), matrix (c(4,6,3,-1,10,7),
4
5 # Provide names to the elements in the list.
6
7 names(list_data) <-c ("1st Quarter", "A_Matrix", "A Inner list")
8
9 # Show the list.
10
11 print(list_data)
12
```

The status bar at the bottom of the editor shows '12:1 (Top Level) R Script'. Below the editor is a console window with the following output:

```
R 4.2.1 ~ /
> list_data <- list(c("March", "April", "June"), matrix (c(4,6,3,-1,10,7), n
row= 2), list("Yellow", 11.2))
> names(list_data) <-c ("1st Quarter", "A_Matrix", "A Inner list")
> print(list_data)
$`1st Quarter`
[1] "March" "April" "June"

$A_Matrix
  [,1] [,2] [,3]
[1,]   4   3  10
[2,]   6  -1   7

$`A Inner list`
$`A Inner list`[[1]]
[1] "Yellow"

$`A Inner list`[[2]]
[1] 11.2
```

Image showing list elements being named

```
# Create a list containing a vector, a matrix and a list.
```

```
list_data <- list(c("March", "April", "June"), matrix (c(4,6,3,-1,10,7), nrow= 2), list("Yellow",  
11.2))
```

```
# Provide names to the elements in the list.
```

```
names(list_data) <-c ("1st Quarter", "A_Matrix", "A Inner list")
```

```
# Show the list.
```

```
print(list_data)
```

Output generated is shown below:

```
$`1st Quarter`  
[1] "March" "April" "June"
```

```
$A_Matrix  
  [,1] [,2] [,3]  
[1,]   4   3  10  
[2,]   6  -1   7
```

```
$`A Inner list`  
$`A Inner list`[[1]]  
[1] "Yellow"
```

```
$`A Inner list`[[2]]  
[1] 11.2
```

Accessing list elements:

Elements can be accessed by the index of the element in the list. In case the lists are named then it can also be accessed using the names.

The screenshot shows the RStudio interface. The top menu bar includes File, Edit, Code, View, Plots, Session, Build, Debug, Profile, Tools, and Help. Below the menu is a toolbar with icons for file operations and a search bar labeled 'Go to file/function'. The main editor window, titled 'Untitled1*', contains the following R code:

```
1 # Create a list containing a vector, a matrix and a list.
2
3 list_data <- list(c("March", "April", "June"), matrix (c(4,6,3,-1,10,7),
4
5 # Provide names to the elements in the list.
6
7 names(list_data) <-c ("1st Quarter", "A_Matrix", "A Inner list")
8
9 # show the list.
10
11 print(list_data)
12
```

The status bar at the bottom of the editor shows '12:1 (Top Level)' and 'R Script'. Below the editor is a console window with the following output:

```
R 4.2.1 ~ /
> list_data <- list(c("March", "April", "June"), matrix (c(4,6,3,-1,10,7), n
row= 2), list("Yellow", 11.2))
> names(list_data) <-c ("1st Quarter", "A_Matrix", "A Inner list")
> print(list_data)
$`1st Quarter`
[1] "March" "April" "June"

$A_Matrix
  [,1] [,2] [,3]
[1,]   4   3  10
[2,]   6  -1   7

$`A Inner list`
$`A Inner list`[[1]]
[1] "Yellow"

$`A Inner list`[[2]]
[1] 11.2
```

Image showing list elements being accessed

```
# Create a list containing a vector, a matrix and a list.
```

```
list_data <- list(c("March", "April", "June"), matrix(c(4,6,3,-1,10,7), nrow= 2), list("Yellow",  
11.2))
```

```
# Provide names to the elements in the list.
```

```
names(list_data) <- c("1st Quarter", "A_Matrix", "A Inner list")
```

```
# Show the list.
```

```
print(list_data)
```

Output:

```
$`1st Quarter`  
[1] "March" "April" "June"
```

```
$A_Matrix  
  [,1] [,2] [,3]  
[1,]   4   3  10  
[2,]   6  -1   7
```

```
$`A Inner list`  
$`A Inner list`[[1]]  
[1] "Yellow"
```

```
$`A Inner list`[[2]]  
[1] 11.2
```

```
# Give names to the elements in the list.
```

```
names(list_data) <- c("1st Quarter", "A_Matrix", "A Inner list")
```

```
# Access the first element of the list.
```

```
print(list_data[1])
```

```
# Access the thrid element. As it is also a list, all its elements will be printed.
```

```
print(list_data[3])
```

```
# Access the list element using the name of the element.
```

```
print(list_data$A_Matrix)
```

When the code is executed the following will be the result displayed:

```
$`1st Quarter`  
[1] "March" "April" "June"  
  
> print(list_data$A_Matrix)  
  [,1] [,2] [,3]  
[1,]  4   3  10  
[2,]  6  -1   7
```

The list elements can also be manipulated:

One can add delete and update list elements. One can add and delete elements only at the end of a list. But one can update any element.

```
# Create a list containing a vector, a matrix and a list.
```

```
list_data <-list(c("Jan", "Feb", "Mar"), matrix (c(2,4,6,2,-5,8), nrow=2),list("blue", 10.2))
```

```
# Give names to the elements in the list.
```

```
names(list_data) <-c("1st Quarter", "A_Matrix", "A Inner list")
```

```
# Add element at the end of the list.
```

```
list_data[4] <- "New element"
```

```
print(list_data[4])
```

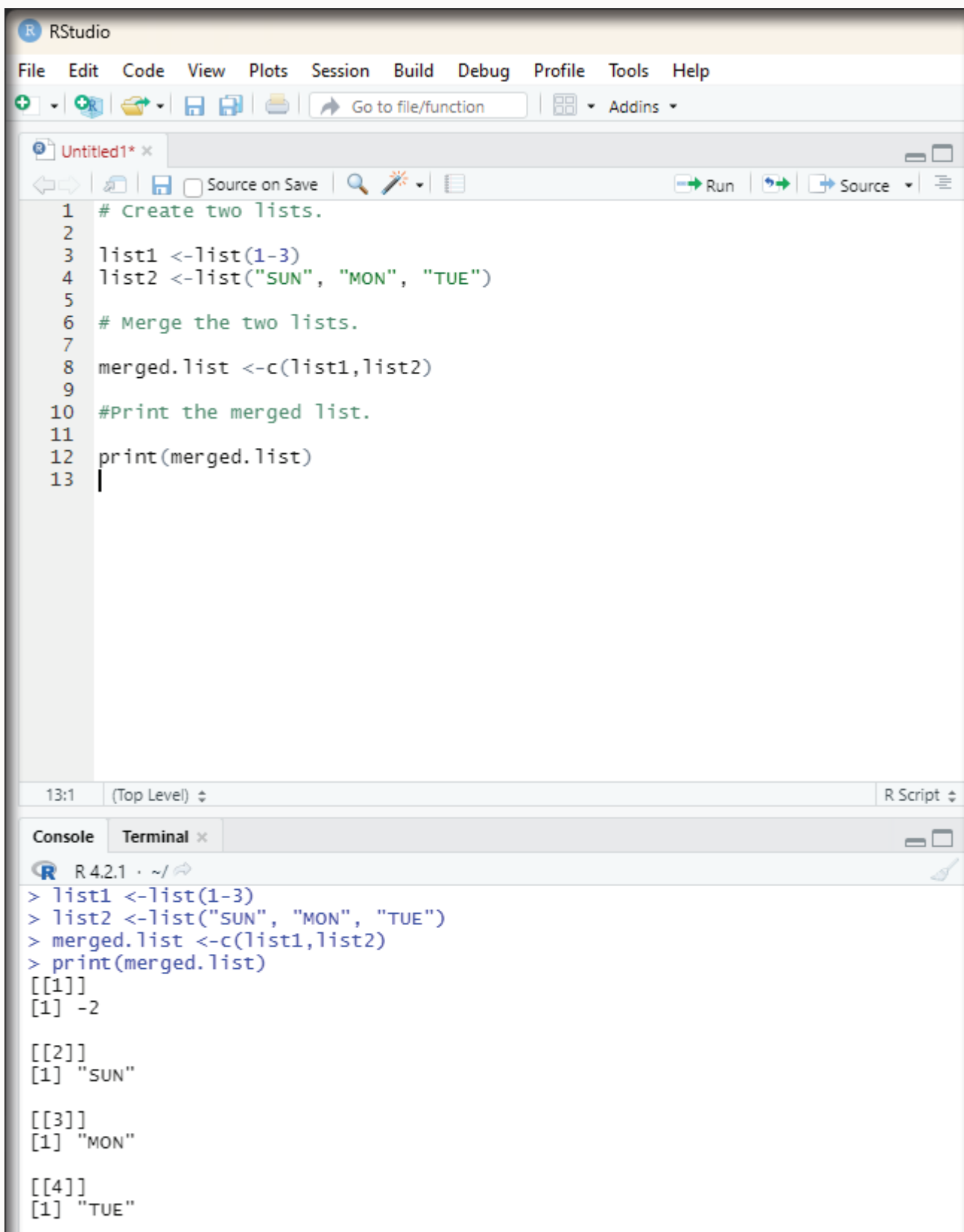
```
# Update the 3rd element
```

```
list_data[3] <- "updated element"
```

```
print(list_data[3])
```

Merging lists:

Lists can be merged into one list by placing all the lists inside one list() function.



The image shows the RStudio interface. The top menu bar includes File, Edit, Code, View, Plots, Session, Build, Debug, Profile, Tools, and Help. Below the menu is a toolbar with icons for file operations and a search bar labeled 'Go to file/function'. The main editor window, titled 'Untitled1*', contains the following R code:

```
1 # Create two lists.
2
3 list1 <-list(1-3)
4 list2 <-list("SUN", "MON", "TUE")
5
6 # Merge the two lists.
7
8 merged.list <-c(list1,list2)
9
10 #Print the merged list.
11
12 print(merged.list)
13
```

Below the editor is a status bar showing '13:1 (Top Level)' and 'R Script'. At the bottom is a console window with the following output:

```
R 4.2.1 ~/  
> list1 <-list(1-3)  
> list2 <-list("SUN", "MON", "TUE")  
> merged.list <-c(list1,list2)  
> print(merged.list)  
[[1]]  
[1] -2  
  
[[2]]  
[1] "SUN"  
  
[[3]]  
[1] "MON"  
  
[[4]]  
[1] "TUE"
```

Image showing how to merge two lists


```
# Create two lists.
```

```
list1 <-list(1-3)
```

```
list2 <-list("SUN", "MON", "TUE")
```

```
# Merge the two lists.
```

```
merged.list <-c(list1,list2)
```

```
#Print the merged list.
```

```
print(merged.list)
```

On running the code the following output will be generated:

```
[[1]]
```

```
[1] 1
```

```
[[2]]
```

```
[1] 2
```

```
[[3]]
```

```
[1] 3
```

```
[[4]]
```

```
[1] "SUN"
```

```
[[5]]
```

```
[1] "MON"
```

```
[[6]]
```

```
[1] "TUE"
```

Converting list to vector:

A list can be converted to a vector so the events of the vector can be used for further manipulation. All the arithmetic operations on vectors can be applied after the list is converted into vectors. In order to make use of this feature one should use the `unlist()` function. It takes the list as the input and produces a vector.

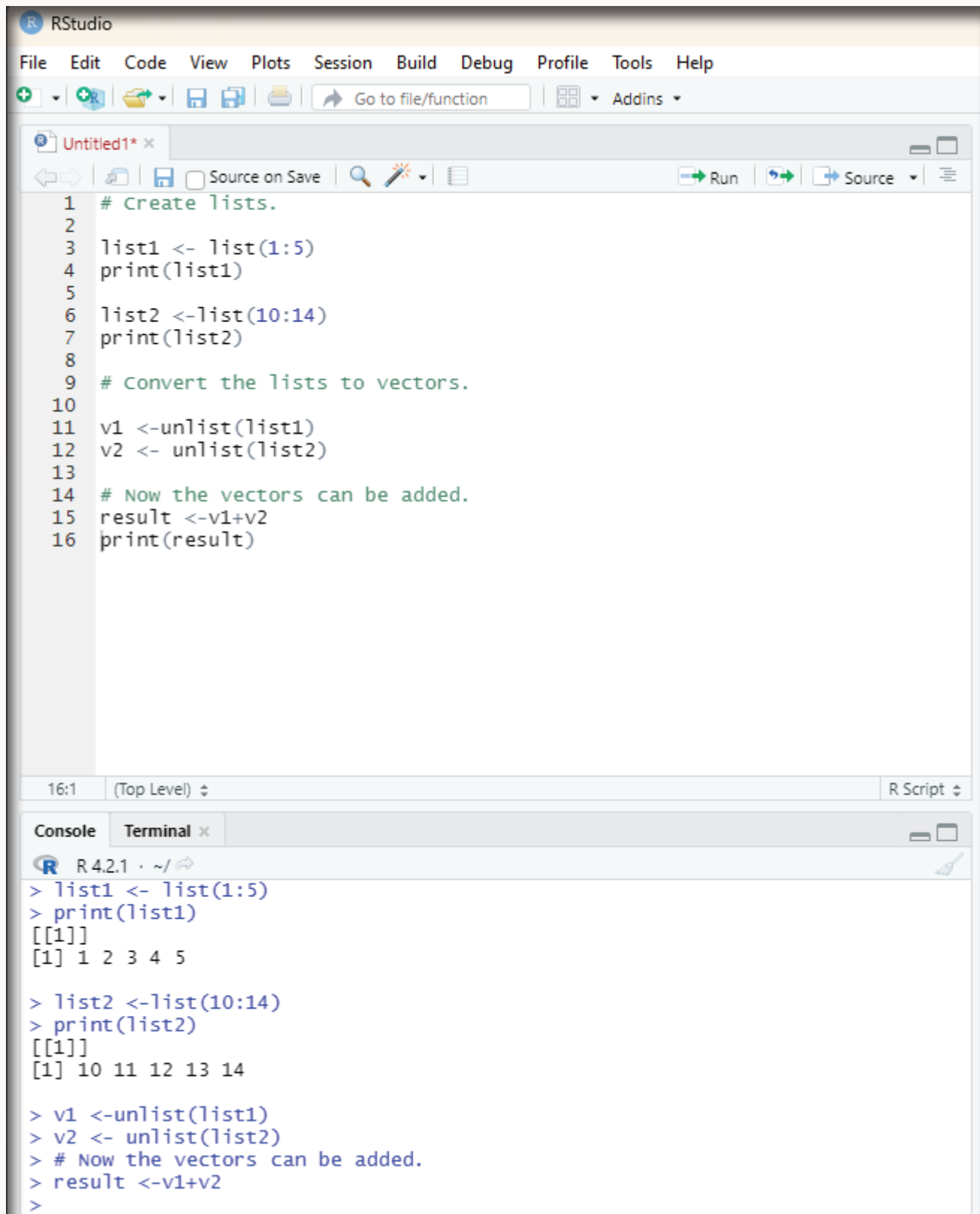


Image showing list converted to vector

```
# Create lists.
```

```
list1 <- list(1:5)  
print(list1)
```

```
list2 <-list(10:14)  
print(list2)
```

```
# Convert the lists to vectors.
```

```
v1 <-unlist(list1)  
v2 <- unlist(list2)
```

```
# Now the vectors can be added.
```

```
result <-v1+v2  
print(result)
```

On running the code the following result will be displayed:

```
[1] 11 13 15 17 19
```

Data frame:

Data frames are data displayed in a format as a table.

Data frames can have different types of data inside it. While the first column can be “character”, the second and third can be “numeric” or “logical”. However, each column should have the same type of data.

This is a table or a two-dimensional array like structure in which each column contains values of one variable and each row contains one set of values from each column.

syntax - data.frame(data)

The following are the characteristics of a data frame:

1. The column names should not be empty
2. The row names should be unique
3. The data stored in a data frame can be of numeric, factor or character type
4. Each column should contain the same number of data items

Example:

The aim is to create a data frame with the following data:

Training

Pulse rate

Duration

Code:

```
# Create a data frame
```

```
Data_Frame <-data.frame (
```

```
  Training = c("Strength", "Stamina", "Other"),  
  Pulse = c(100,150, 120),  
  Duration = c(60,30,45)  
)
```

```
#Print the data frame
```

```
Data_Frame
```

Output:

Training Pulse Duration

```
1 Strength 100    60  
2 Stamina 150    30  
3  Other 120    45
```

In order to get summary of the data the following code can be used:

```
output <-summary(Data_Frame)  
> print(output)
```

Output:

```
Training      Pulse      Duration  
Length:3      Min. :100.0 Min. :30.0  
Class :character 1st Qu.:110.0 1st Qu.:37.5  
Mode :character Median :120.0 Median :45.0  
      Mean :123.3 Mean :45.0  
      3rd Qu.:135.0 3rd Qu.:52.5  
      Max. :150.0 Max. :60.0
```

The image shows the RStudio interface. The top menu bar includes File, Edit, Code, View, Plots, Session, Build, Debug, Profile, Tools, and Help. Below the menu is a toolbar with icons for file operations and a search bar labeled 'Go to file/function'. The main editor window, titled 'Untitled1*', contains the following R code:

```
1 # Create a data frame
2
3 Data_Frame <-data.frame (
4   Training = c("Strength", "Stamina", "other"),
5   Pulse = c(100,150, 120),
6   Duration = c(60,30,45)
7 )
8
9
10 #Print the data frame
11
12 Data_Frame
13 |
```

Below the editor is a status bar showing '13:1' and '(Top Level)'. The bottom panel has tabs for 'Console' and 'Terminal'. The 'Console' tab is active, showing the execution of the code and the resulting data frame:

```
> Data_Frame <-data.frame (
+
+   Training = c("Strength", "Stamina", "other"),
+   Pulse = c(100,150, 120),
+   Duration = c(60,30,45)
+ )
> Data_Frame
  Training Pulse Duration
1 Strength  100      60
2 Stamina  150      30
3  other   120      45
```

Image showing data frame being created using RStudio

Access items from data frame:

Items from data frame can be accessed using [] single brackets, [[]] double brackets, or \$ symbol.

Example:

```
Data_Frame<- data.frame(  
  Training = c("strength", "stamina", "other"),  
  Pulsee = c(100, 150, 120),  
  Duration = c(60,30,45)  
)
```

```
Data_Frame [1]
```

```
Data_Frame[["Training"]]
```

```
Data_Frame$Training
```

Code:

```
# Create the data frame.
```

```
emp.data <-data.frame(  
  emp_id = c (1:5),  
  emp_name = c("John", "Murphy", "Sundar", "Ramesh", "Bony"),  
  salary = c(600, 528.49, 789,854.8, 658),  
  start_date = as.Date (c("2012-05-06", "2012-06-22", "2013-03-22", "2015-04-16","2016-02-1")),
```

```
stringsAsFactors = FALSE
```

```
)
```

```
# Print/display data frame
```

```
print(emp.data)
```

Structure of the data frame can be seen by using str() function.

```
str(emp.data)
```

Summary of Data in Data Frame:

The statistical summary and nature of the data can be obtained by applying summary() function.

Data can be extracted from Data Frame by using column name.

```
#Extract Specific Columns.
```

```
result <-data.frame(emp.data$emp_name, emp.data$salary)
```

The user can also extract the first two rows and then all columns.

Code:

```
#Extract first two rows.
```

```
result <- emp.data[1:2,]
```

```
print (result)
```

```
print(result)
```

```
# Extract 3rd and 5th row with 2nd and 4th column.
```

```
result <- emp.data[c(3,5),c(2,4)]
```

```
print(result)
```

One can Expand Data Frame by adding columns and rows. code for adding column:

```
#Add the"dept" column
```

```
emp.data$dept <- c("IT","Operations", "IT", "HR", "Finance")
```

```
v <-emp.data
```

```
print (v)
```

Adding rows to the existing data frame:

To add more rows permanently to an existing data frame, one needs to bring in new rows in the same structure as the existing data frame. For this purpose rbind() function can be used.

Adding Rows using rbind() function:

Example:

```
Data_Frame <-data.frame(
```

```
Training = c("Strength", "Stamina", "Other"),
```

```
Pulse = c(100,150,120),
```

```
Duration = c(60,30,45))
```

```
# Add a new row.
```

```
New_row_DF <- rbind(Data_Frame, c("Strength", 110, 110))
```

```
# Print the new row
```

```
New_row_DF
```

Add columns:

Extra columns can be added using cbind() function in a data frame.

Example:

```
Data_Frame <- data.frame(
```

```
  Training = c("Strength", "Stamina", "other"),
```

```
  Pulse = c(100, 150, 120),
```

```
  Duration = c(60,35,40)
```

```
)
```

```
#Add a new column:
```

```
New_col_DF <- cbind (Data_Frame, Steps = c(1000,6000,2000))
```

```
# Print the new column.
```

```
New_col_DF
```

Output:

	Training	Pulse	Duration	Steps
--	----------	-------	----------	-------

1	Strength	100	60	1000
---	----------	-----	----	------

2	Stamina	150	35	6000
---	---------	-----	----	------

3	other	120	40	2000
---	-------	-----	----	------

```
# Create the second data frame
```

```
emp.newdata <- data.frame(
```

```
  emp_id = c (6:8),
```

```
  emp_name = c("Kumar","kurnal","Abhay"),
```

```
  salary = c(578.0,722.5,632.8),
```

```
  start_date = as.Date(c("2013-05-21","2013-07-30","2014-06-17")),
```

```
  dept = c("IT","Operations","Fianance"),
```

```
  stringsAsFactors = FALSE
```

```
)
```



```
# Bind the two data frames.
emp.finaldata <- rbind(emp.data,emp.newdata)
print(emp.finaldata)
```

Removing rows and columns in a Data Frame:

In order to remove rows and columns c() function can be used.

Example:

```
Data_Frame <- data.frame (
  Training = c("Strength", "Stamina", "Other"),
  Pulse = c(100,130,120),
  Duration = c(60,30,20)
)

# Remove the first row and column.

Data_frame_new <- Data_Frame[-c(1), -c(1)]

# Print the new data frame

Data_frame_new
```

Output:

```
Pulse Duration
2  130      30
3  120      20
```

Amount of Rows and Columns in Data frame:

Amount of rows and columns in a Data frame can be ascertained using dim() function.

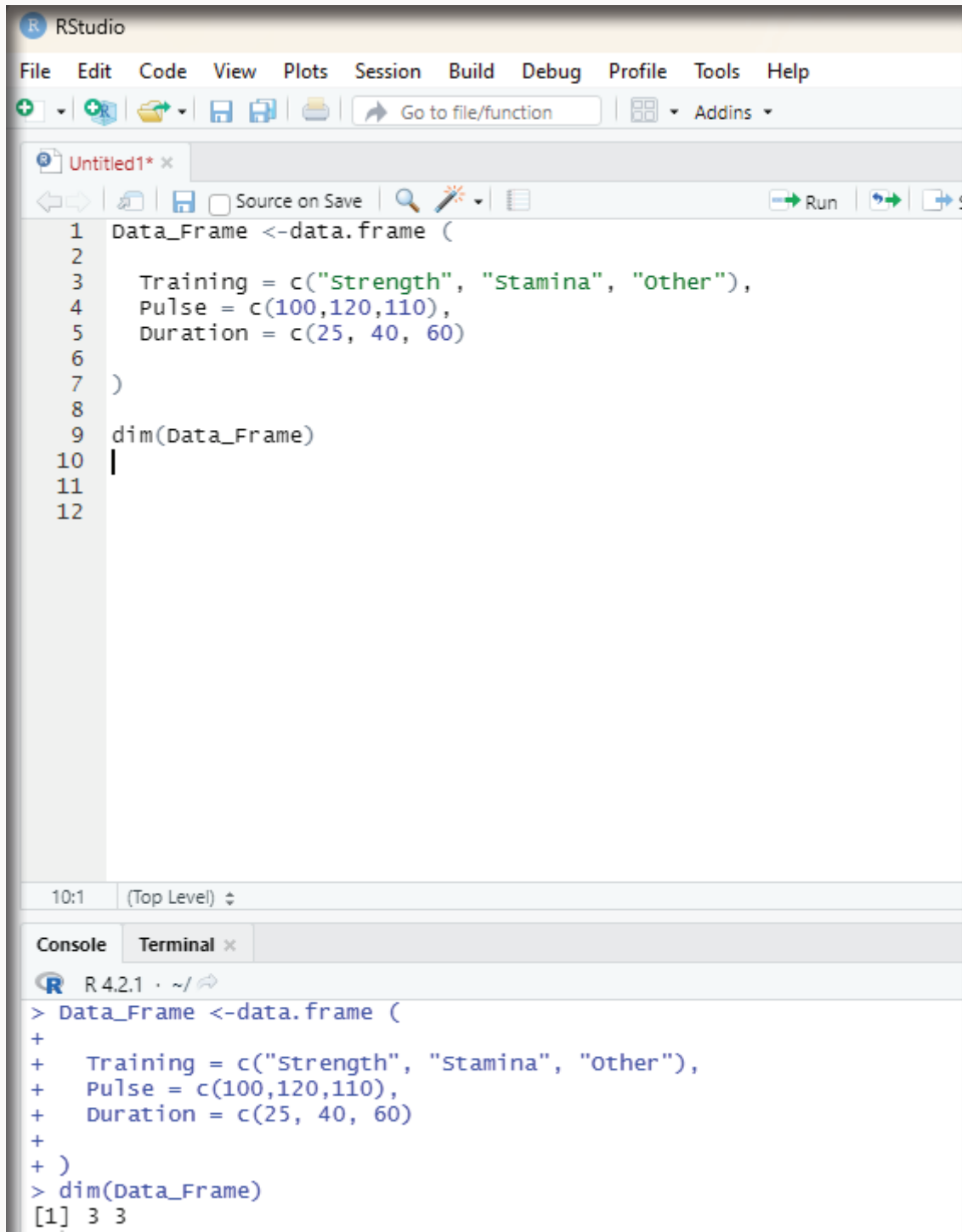
Example:

```
Data_Frame <- data.frame (
  Training = c("Strength", "Stamina", "Other"),
  Pulse = c(100,120,110),
  Duration = c(25, 40, 60)
)

dim(Data_Frame)
```

Output:

```
[1] 3 3
```



The screenshot shows the RStudio interface. The source editor contains the following R code:

```
1 Data_Frame <-data.frame (  
2  
3   Training = c("Strength", "Stamina", "Other"),  
4   Pulse = c(100,120,110),  
5   Duration = c(25, 40, 60)  
6  
7 )  
8  
9 dim(Data_Frame)  
10 |  
11  
12
```

The console at the bottom shows the execution of the code and the resulting output:

```
> Data_Frame <-data.frame (  
+   Training = c("Strength", "Stamina", "Other"),  
+   Pulse = c(100,120,110),  
+   Duration = c(25, 40, 60)  
+ )  
> dim(Data_Frame)  
[1] 3 3
```

Image showing estimating the number of rows and columns using RStudio

One can also use the `ncol()` function to find the number of columns and `nrow()` function to find the number of rows.

```
ncol(Data_Frame)
```

```
nrow(Data_Frame)
```

Output:

```
> ncol(Data_Frame)
```

```
[1] 3
```

```
>
```

```
> nrow(Data_Frame)
```

```
[1] 3
```

Data Frame Length:

In order to ascertain the number of columns in a Data frame `length()` function can be used (similar to `ncol()` function).

```
length(Data_Frame)
```

Output:

```
length(Data_Frame)
```

```
[1] 3
```

R does not have a spread sheet type of data entry facility. (Something similar to that of Excel). There are ways to invoke a spreadsheet like data entry tool in R.

First step:

Object must be created. Everything in R is considered to be an object and this is actually the fundamental distinction between R and Excel. While one can launch a spreadsheet like viewer for data entry in R, one needs to pass the data into an object. In order to do this a blank data frame needs to be setup with rows and columns. If the user leaves the arguments blank in `data.frame()` it would result in an empty data frame.

```
myData<- data.frame()
```

Second step:

Data is edited in the viewer.

One has to use the `edit` function to launch the viewer. The user should pass the `myData` data frame back to the `myData` object. In this way the changes made to the module will be saved to the original object.

myData <-(myData)

myData <- edit(myData)

The variable names can be changed by clicking on their labels and typing the changes. One can also set variables as numeric or character.

Note - One cannot set a variable to logical; and it has to be done in the syntax editor.

On data being entered, they get saved automatically.

Third step:

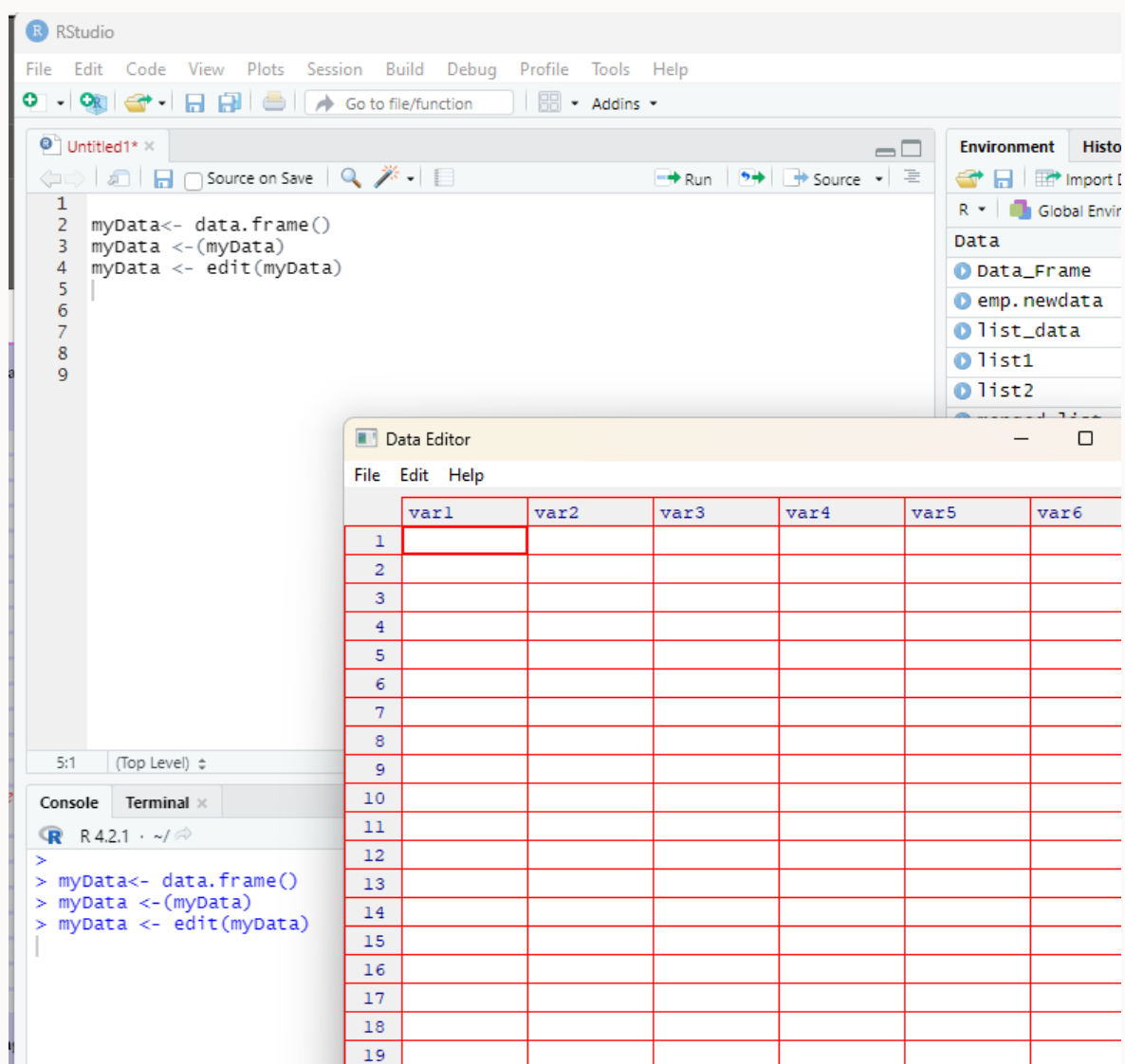


Image showing Data Editor window opening

Data Entry in the spreadsheet format:

In order to change the header name the user needs to click on it. Input window will open prompting the user to key in a new name for the chosen column. The type of data that needs to be entered can also be chosen from this input window. The user has the option of choosing between character and Numerical formats.

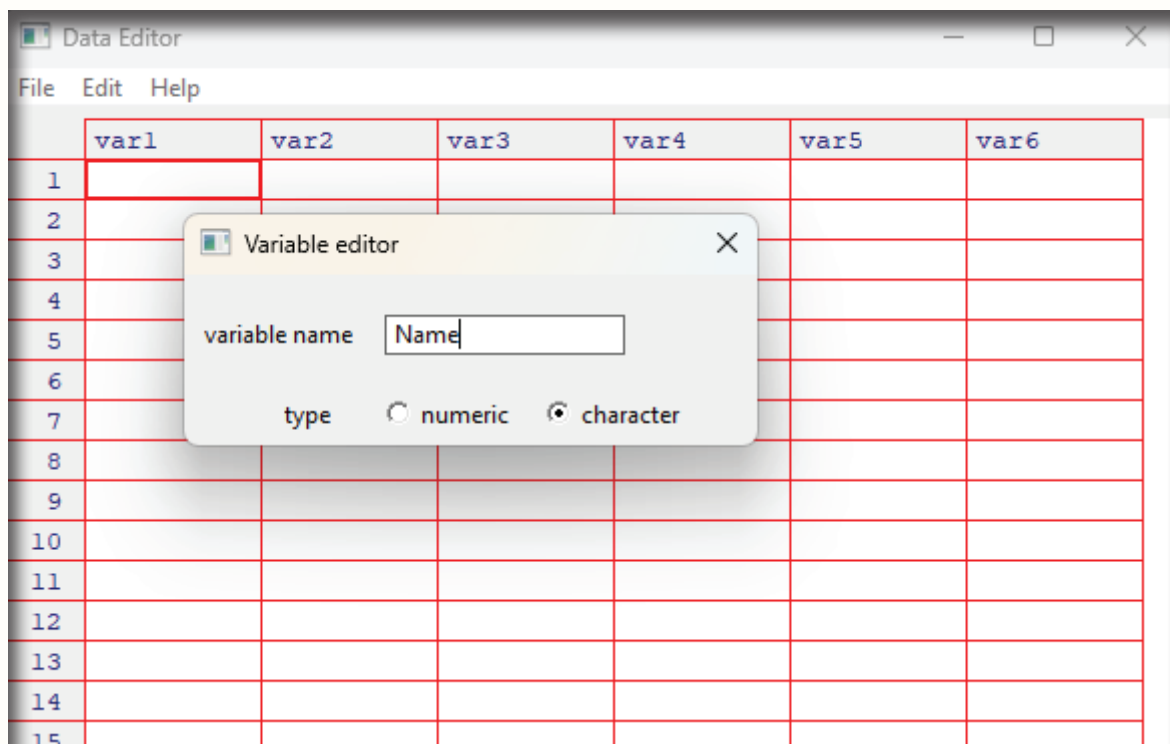


Image showing the variable editor input window that appears on clicking the header of the column. In this image in the variable name column the desired value is entered. In the type of data the desired type of data is also chosen (numeric and character).

Variable editor does not provide the option of naming the data type as logical. This needs to be done at the level of syntax editor using the following command:

```
myData  
is.logical(myData$IsInjured)  
myData$IsInjured <- as.logical(myData$IsInjured)
```

This syntax is specifically for the example given. The user can change the name of the data in the syntax before executing. This example is provided with an intention that the user should familiarize themselves with various syntax that can be used in R.

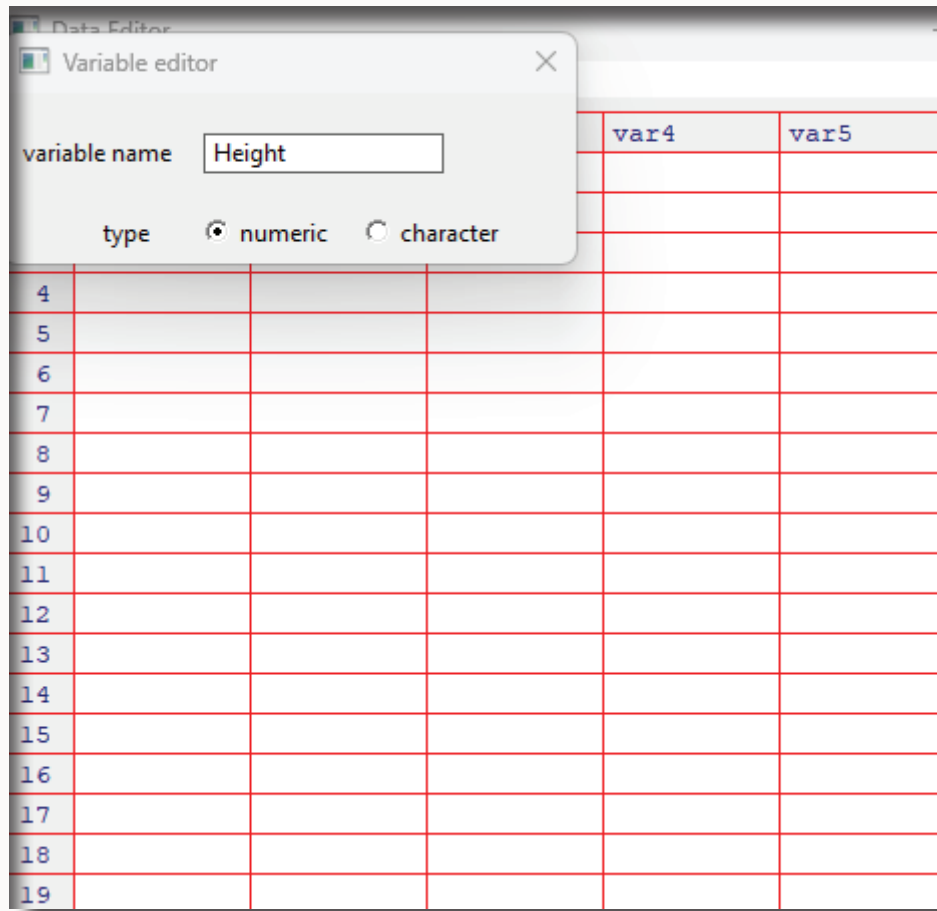


Image showing second variable name being changed to Height and the type of data that is to be entered in this column is chosen as numeric.

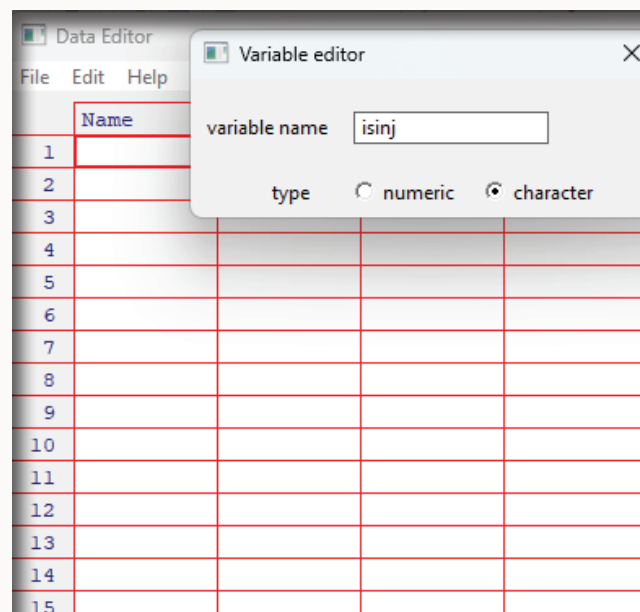
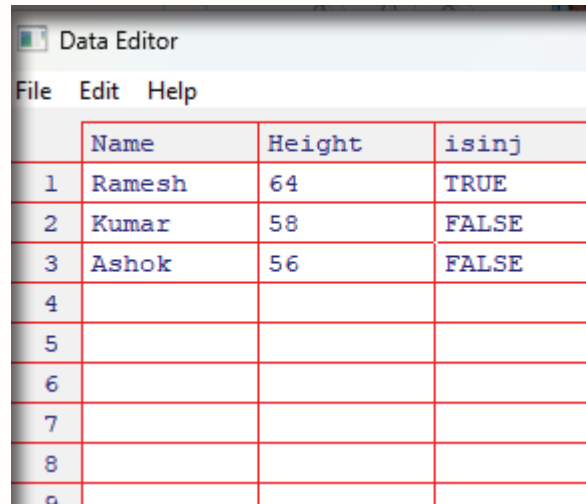


Image showing the third variable name changed to reflect the status whether injured or not. Type of data eventhough it is logical cannot be specified here. Only character needs to be choosen.

Data can be entered in each of these columns as shown below.



	Name	Height	isinj
1	Ramesh	64	TRUE
2	Kumar	58	FALSE
3	Ashok	56	FALSE
4			
5			
6			
7			
8			
9			

When the table is closed it automatically gets saved.

As stated earlier the data editor does not set the columns to logical. It can be assigned only using the syntax editor.

Code for setting the columns as logical:

```
myData
is.logical(myData$IsInjured)
myData$IsInjured <- as.logical(myData$IsInjured)
```

Full code:

```
#create blank data frame
myData <- data.frame()
#edit data in the viewer
myData <- edit(myData)
#close & load
myData
#change IsInjured to Logical
is.logical(myData$IsInjured)
myData$IsInjured <- as.logical(myData$IsInjured)
```

Operators in R Programming

Operators are symbols that tell the compiler to perform specific mathematical or logical computations. R language is rich in built-in operators and provides the following types of operators:

1. Arithmetic operators
2. Relational operators
3. Logical operators
4. Assignment operators
5. Miscellaneous operators

Arithmetic operators:

These operators are used to perform arithmetic calculations. They include:

+ Adds two vectors

- Subtracts the second vector from the first

*** Multiplies both vectors**

/ Divide the first vector with the second

%% Divide the first vector with the second and display the remainder

%%/ It provides the result of division of first vector with second one (quotient).

^ The first vector is raised to the exponent of second vector.

In this example two vectors `v` and `x` are created holding a series of numbers. The intention is to add the numbers in the first vector (`v`) with that of the second (`x`) and display the result.

```
print(m)
```

Prof. Dr Balasubramanian Thiagarajan

Output:

```
[1] 3 9 11 9
```

Subtraction:

In this example two vectors `v` and `x` are created holding a series of numbers. The intention is to subtract the numbers in the second vector `x` from the first vector `v` and display the result.

Code:

(`c` next to `=` sign is an assignment operator. It will be discussed later under assignment operators.

```
v=c(2,4,5,7)
```

```
x=c(1,2,3,2)
```

```
m = v-x
```

```
print(m)
```

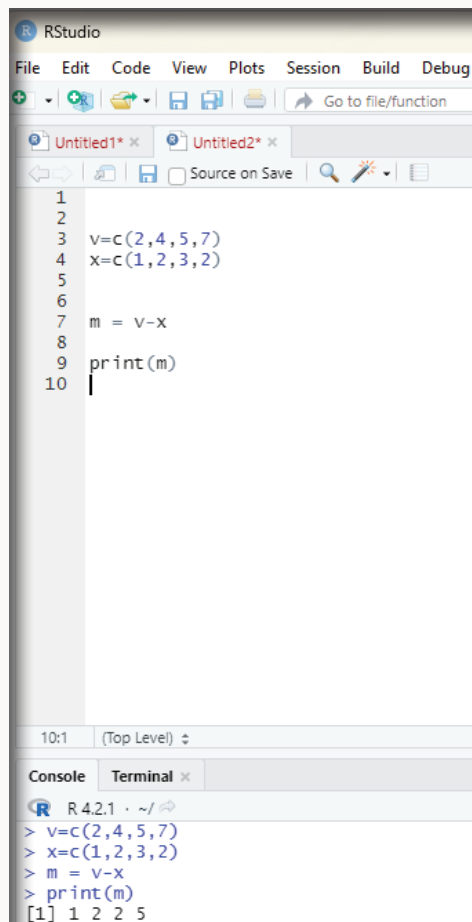


Image showing the subtraction code executed in RStudio

Output:

```
[1] 1 2 2 5
```

Multiplication operator:

* - Multiplies both vectors

Example:

```
v = c(2,4,6,8)
```

```
s = c(2,5,6,1)
```

```
m = (v*s)
```

```
print(m)
```

Output generated on running the code:

```
[1] 4 20 36 8
```

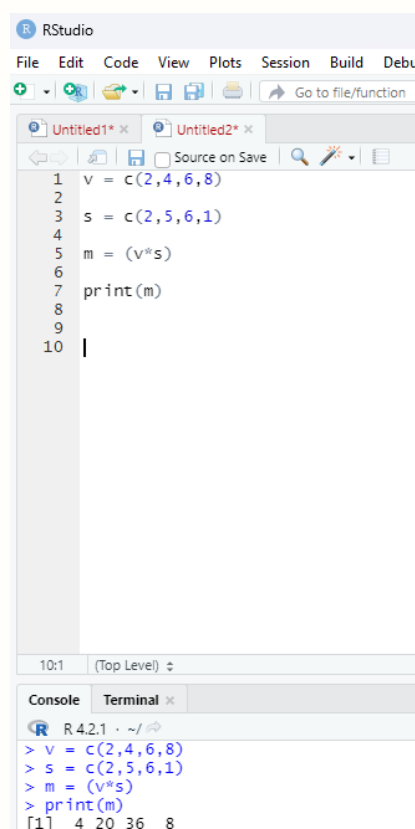


Image showing multiplication operator in use

Division Operator:

Division operator:

/ - This operator divides the first vector with the second.

```
# Create two vectors with four numbers each.
```

```
x = c(2,5,4,34)
```

```
y = c(1,5,3,12)
```

```
z = (x/y)
```

```
print(z)
```

Output:

```
2.000000 1.000000 1.333333 2.833333
```

Dividing the first vector with the second vector and displaying only the remainder.

The operator used for this purpose is %%.

Example:

In this example two variables x and y are created. Numerical values are assigned to each of these variables. The first variable x is divided with the second variable y. The remainder is displayed if %% operator is used.

Code:

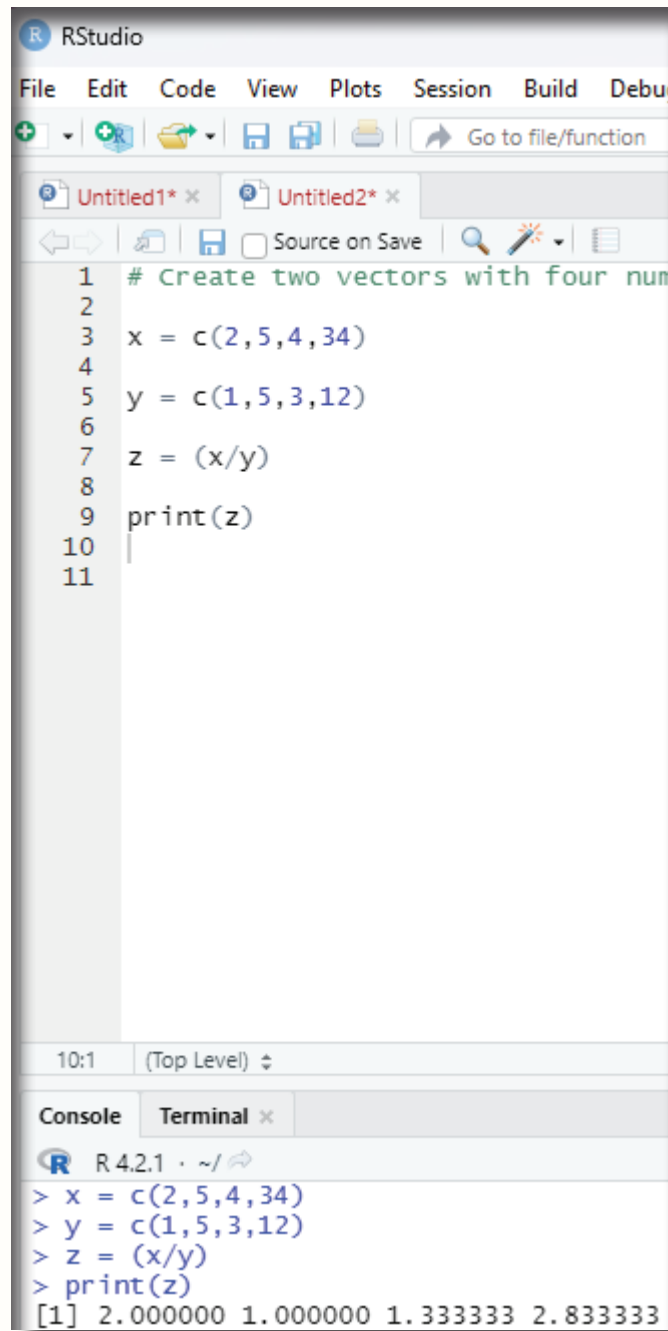
```
x = 5
```

```
y = 2
```

```
print(x%%y)
```

Output:

```
[1] 1
```



The image shows the RStudio interface. The top menu bar includes File, Edit, Code, View, Plots, Session, Build, and Debug. Below the menu is a toolbar with icons for file operations and a 'Go to file/function' search bar. The editor pane shows two untitled files. The first file contains the following R code:

```
1 # Create two vectors with four num
2
3 x = c(2,5,4,34)
4
5 y = c(1,5,3,12)
6
7 z = (x/y)
8
9 print(z)
10
11
```

The console pane at the bottom shows the execution of the code:

```
R 4.2.1 ~/  
> x = c(2,5,4,34)  
> y = c(1,5,3,12)  
> z = (x/y)  
> print(z)  
[1] 2.000000 1.000000 1.333333 2.833333
```

Image showing division operator being used

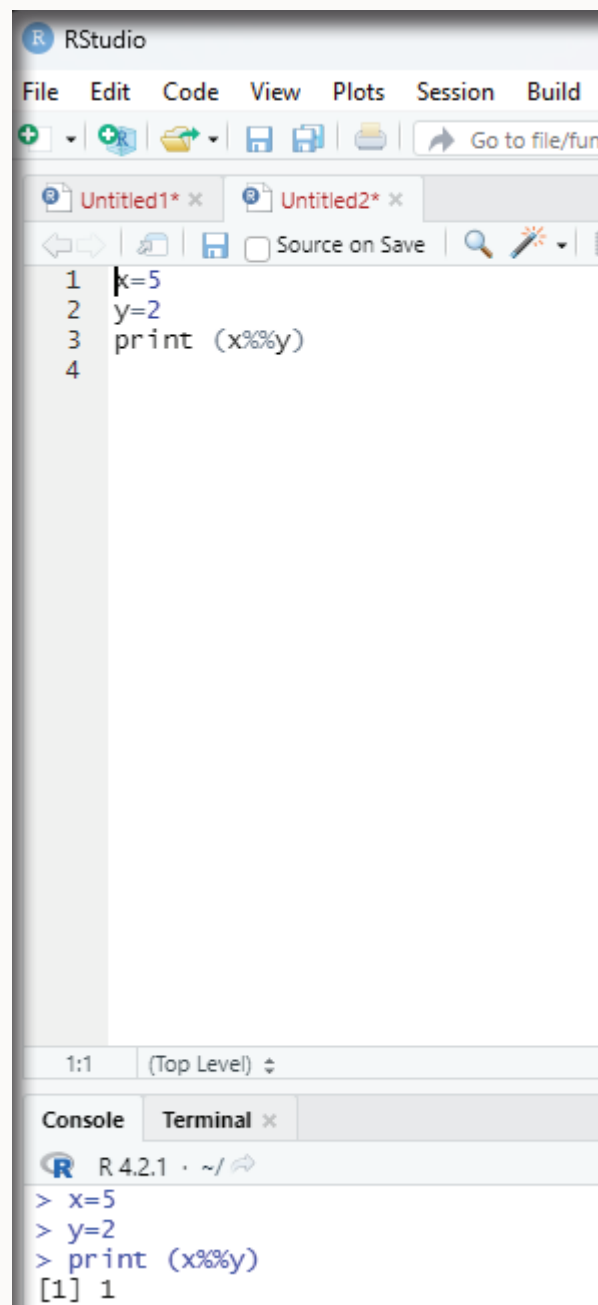


Image showing division operator with display of remainder

Example showing the role of %% in vectors containing a number of numeric variables.

Code:

```
x=c(5, 3, 4, 6)
```

```
y=c( 2, 2, 3, 2)
```

```
print (x%%y)
```

Output:

```
1 1 1 0
```

Code for the result of division of first vector with that of second. Displaying only the quotient and not the remainder.

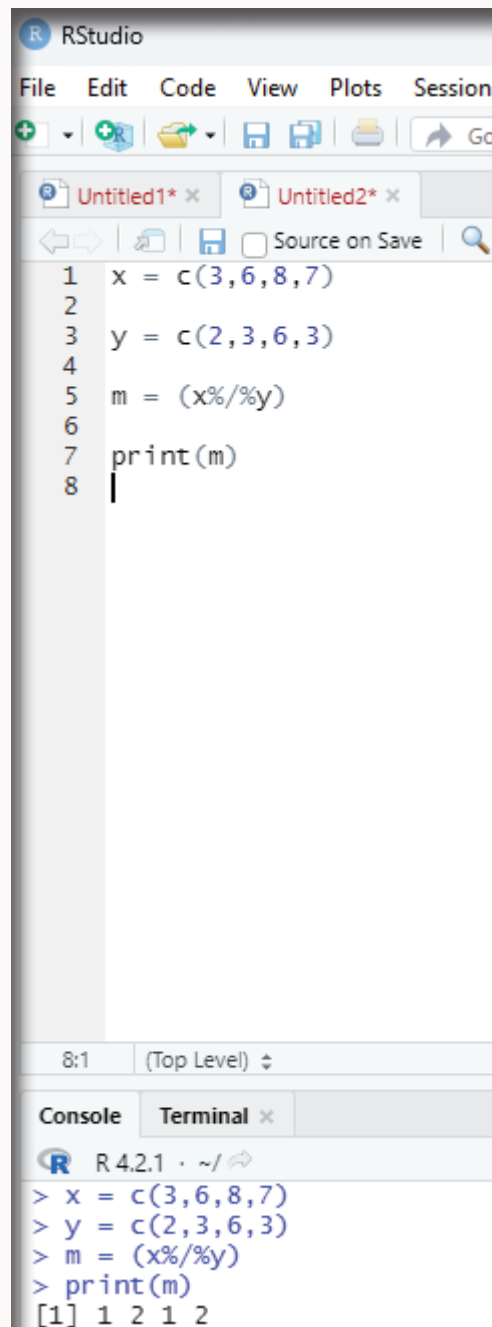
```
x = c(3,6,8,7)
```

```
y = c(2,3,6,3)
```

```
m = (x%%/y)
```

```
print(m)
```

Output: 1 2 1 2



The image shows the RStudio interface. The source editor contains the following R code:

```
1 x = c(3,6,8,7)
2
3 y = c(2,3,6,3)
4
5 m = (x%%y)
6
7 print(m)
8
```

The console at the bottom shows the execution of these commands:

```
> x = c(3,6,8,7)
> y = c(2,3,6,3)
> m = (x%%y)
> print(m)
[1] 1 2 1 2
```

Image showing quotient being displayed after performing the division. The operator used is %/%

Exponent operator: (^)

Exponent is defined as the number of times a number is multiplied by itself.

Example 2 to the third exponent means $2 \times 2 \times 2 = 8$.

Code:

```
x = c(2,5,5,6)
```

```
y = c(2,3,4,2)
```

```
z = (x^y)
```

```
print(z)
```

Output:

```
4 125 625 36
```

Relational operators:

In this each element of the first vector is compared with that of the corresponding element of the second vector. the result of this comparison is a Boolean value. Given below are the list of various relational operators.

> Checks if each element of the first vector is greater than the corresponding element of the second vector.

< Checks if each element of the first vector is less than the corresponding element of second vector.

== Checks if each element of the first vector is equal to the corresponding element of the second vector.

<= Checks if each element of the first vector is less than or equal to the corresponding element of the second vector.

>= Checks if each element of the first vector is greater than or equal to the corresponding element of the second vector.

!= Checks if each element of the first vector is unequal to the corresponding element of second vector.

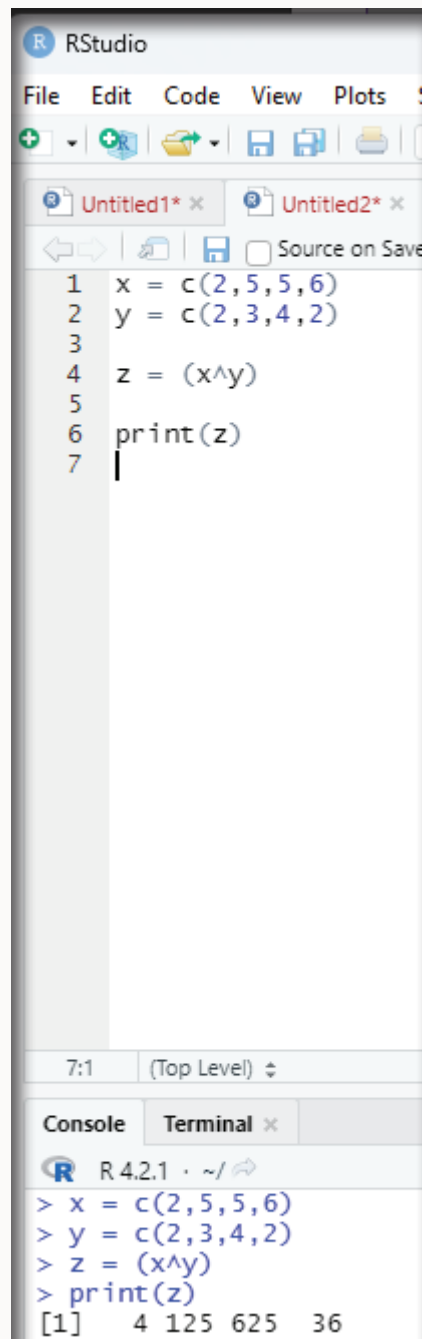


Image showing the use of exponent operator in R programming

Logical operators: These are symbol / word used to connect two or more expressions such that the value of the compound expression produced depends only on that of the original expressions and on the meaning of the operator. Common logical operators include AND, OR and NOT.

$\&$ - It is known as Element-wise Logical AND operator. It combines each element of the first vector with that of the corresponding element of the second vector and gives a output TRUE if both the elements are TRUE.

| - It is called Element-wise Logical OR operator. It combines each element of the first vector with the corresponding element of the second vector and gives a output TRUE if one of the elements is TRUE.

! - It is known as Logical NOT operator. It takes each element of the vector and gives the opposite logical value.

$\&\&$ - It is called logical AND operator. It takes the first element of both the vectors and gives the TRUE only if both are TRUE.

|| - It is called Logical OR operator. It takes the first element of both the vectors and gives the TRUE if one of them is TRUE.

Example for > (greater than):

Code:

```
x = c(4,6,8,9)
y = c(3,5,7,9)
```

```
print(x>y)
```

Output:

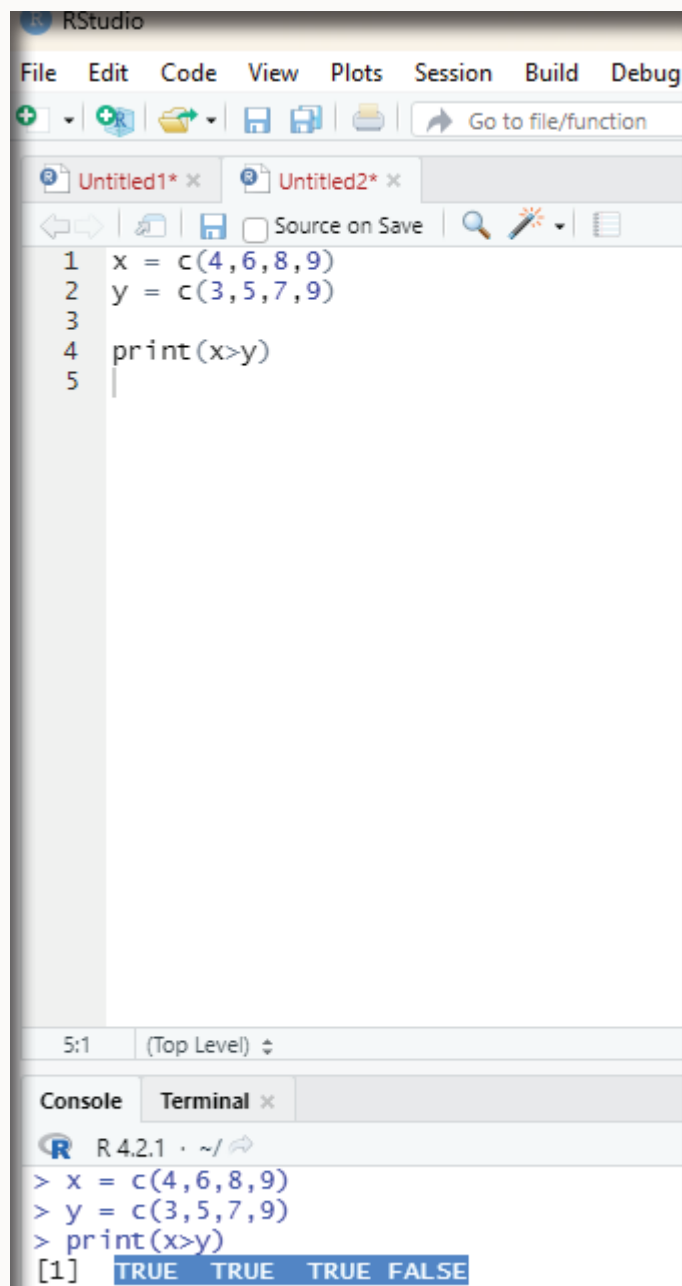
```
TRUE TRUE TRUE FALSE
```

Output reveals that first element of first vector is greater than the first element of second vector - hence the value TRUE.

The second element of first vector is greater than the second element of second vector - hence the value TRUE

The third element of first vector is greater than the third element of second vector - hence the value TRUE.

The fourth element of first vector is less than the fourth element of second vector - hence the value FALSE.



The image shows the RStudio interface. The source editor contains the following R code:

```
1 x = c(4,6,8,9)
2 y = c(3,5,7,9)
3
4 print(x>y)
5
```

The console at the bottom shows the execution of the code:

```
R 4.2.1 ~/  
> x = c(4,6,8,9)  
> y = c(3,5,7,9)  
> print(x>y)  
[1] TRUE TRUE TRUE FALSE
```

Image showing the use of > (greater than operator)

Example for < (lesser than):

Code:

Code:

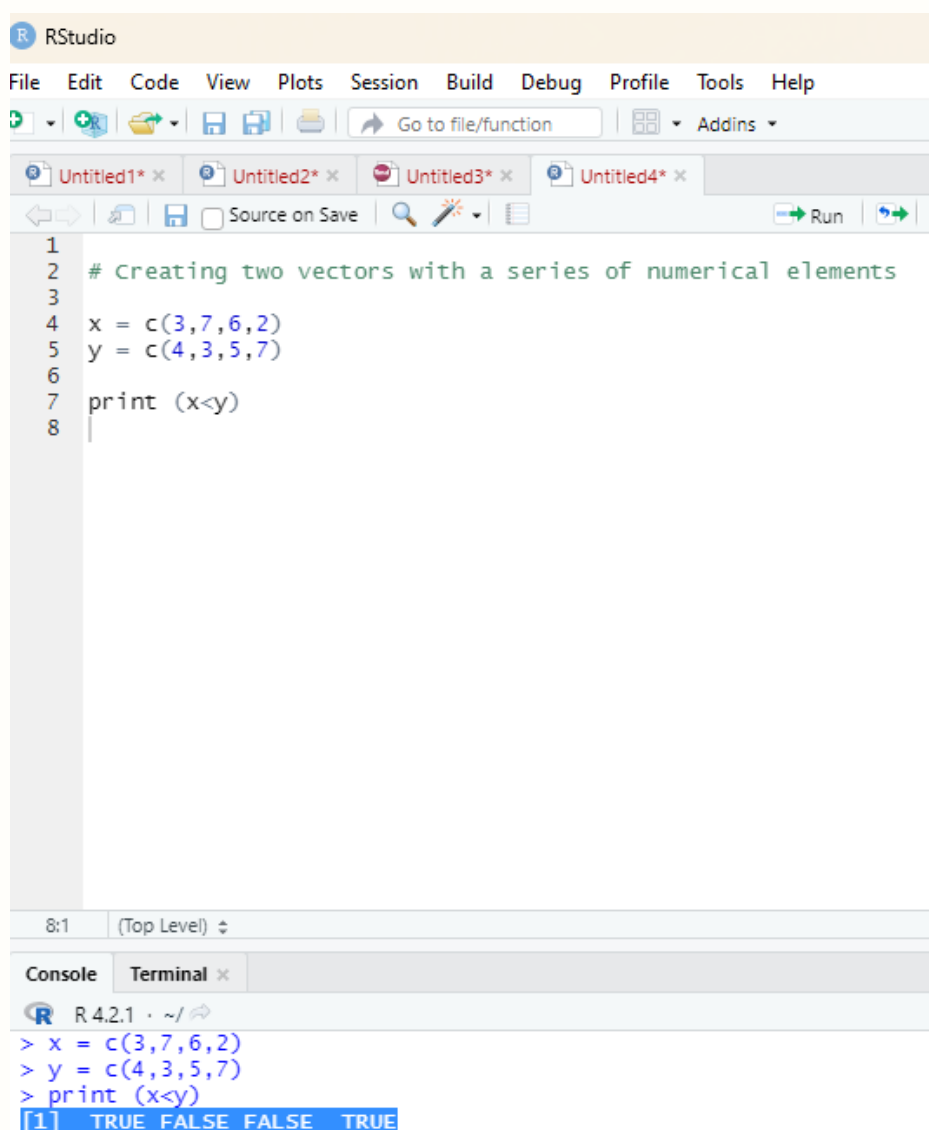
```
x = c(3,7,6,2)
```

```
y = c(4,3,5,7)
```

```
print (x<y)
```

Output:

```
[1] TRUE FALSE FALSE TRUE
```



The screenshot shows the RStudio interface. The script editor contains the following code:

```
1  
2 # Creating two vectors with a series of numerical elements  
3  
4 x = c(3,7,6,2)  
5 y = c(4,3,5,7)  
6  
7 print (x<y)  
8
```

The console at the bottom shows the execution of the code:

```
> x = c(3,7,6,2)  
> y = c(4,3,5,7)  
> print (x<y)  
[1] TRUE FALSE FALSE TRUE
```

Study of output reveals :

The first element of vector x is less than that of the first element of vector y. Hence the value TRUE is printed.

Check to find if each element of the first vector is equal to the corresponding element of the second vector:

Operator - ==

Code:

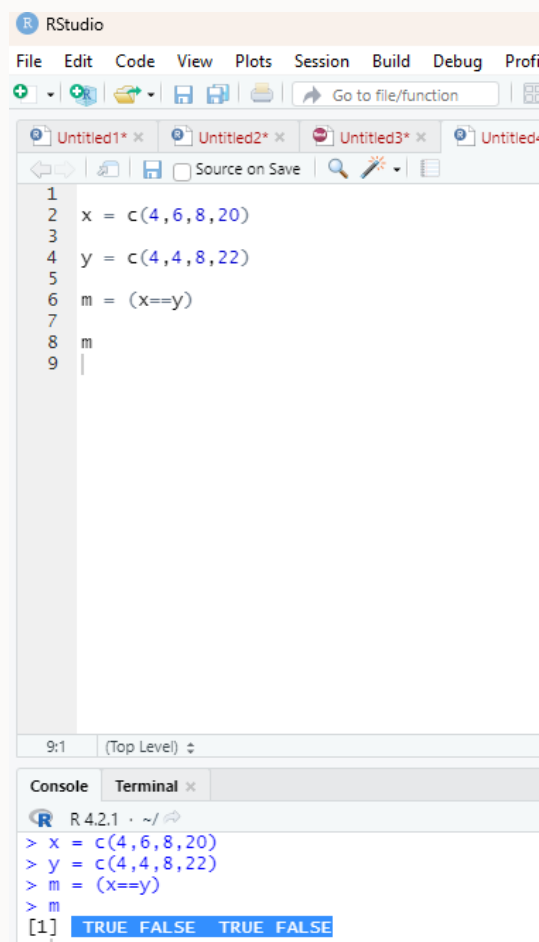
```
x = c(4,6,8,20)
```

```
y = c(4,4,8,22)
```

```
m = (x==y)
```

```
m
```

Output: TRUE FALSE TRUE FALSE



The screenshot shows the RStudio interface. The script editor contains the following code:

```
1  
2 x = c(4,6,8,20)  
3  
4 y = c(4,4,8,22)  
5  
6 m = (x==y)  
7  
8 m  
9
```

The console shows the execution of the code:

```
R 4.2.1 ~/  
> x = c(4,6,8,20)  
> y = c(4,4,8,22)  
> m = (x==y)  
> m  
[1] TRUE FALSE TRUE FALSE
```

Image showing the use of == operator

Operator that is used to check if each element of the first vector is less than or equal to the corresponding element of the second vector:

Operator used:

`<=`

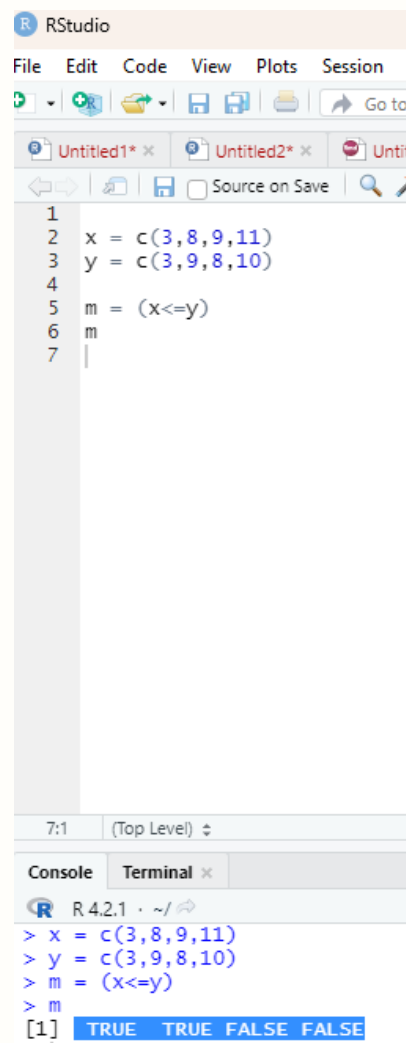
Code:

```
x = c(3,8,9,11)
```

```
y = c(3,9,8,10)
```

```
m = (x<=y)
```

Output : TRUE TRUE FALSE FALSE



The screenshot shows the RStudio interface. The script editor contains the following code:

```
1  
2 x = c(3,8,9,11)  
3 y = c(3,9,8,10)  
4  
5 m = (x<=y)  
6 m  
7 |
```

The console output is as follows:

```
R 4.2.1 ~/  
> x = c(3,8,9,11)  
> y = c(3,9,8,10)  
> m = (x<=y)  
> m  
[1] TRUE TRUE FALSE FALSE
```

Image showing the use of `<=` operator

Operator to check if each element of the first vector is greater than or equal to the corresponding element of the second vector.

Operator used:

`>=`

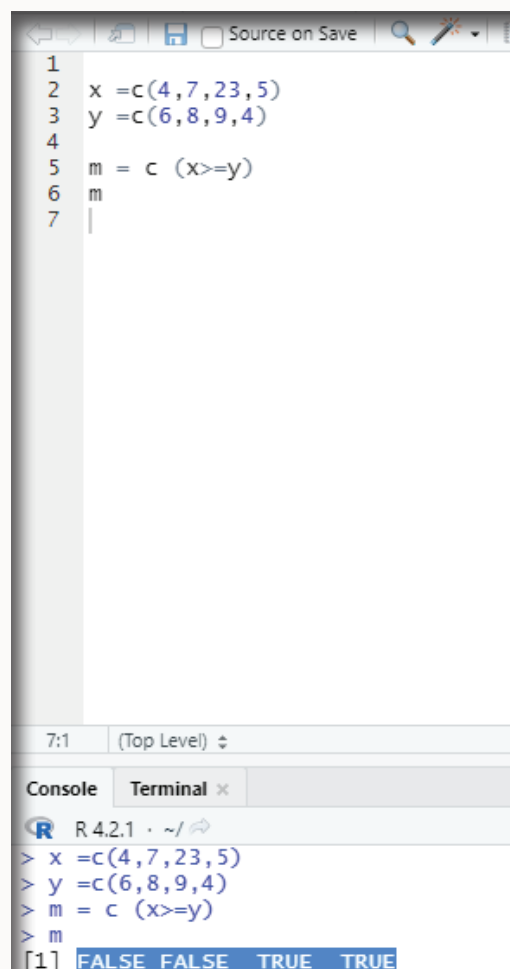
Code:

```
x=c(4,7,23,5)
```

```
y=c(6,8,9,4)
```

```
m = c (x>=y)
```

Output : FALSE FALSE TRUE TRUE



```
1
2 x =c(4,7,23,5)
3 y =c(6,8,9,4)
4
5 m = c (x>=y)
6 m
7 |
```

7:1 (Top Level) ↕

Console Terminal x

R 4.2.1 · ~/

```
> x =c(4,7,23,5)
> y =c(6,8,9,4)
> m = c (x>=y)
> m
[1] FALSE FALSE TRUE TRUE
```

Image showing the use of `>=` operator

Operator to check if each element of the first vector is unequal to the corresponding element of the second vector.

Operator:

`!=`

Code:

```
x=c(4,7,8,9)
```

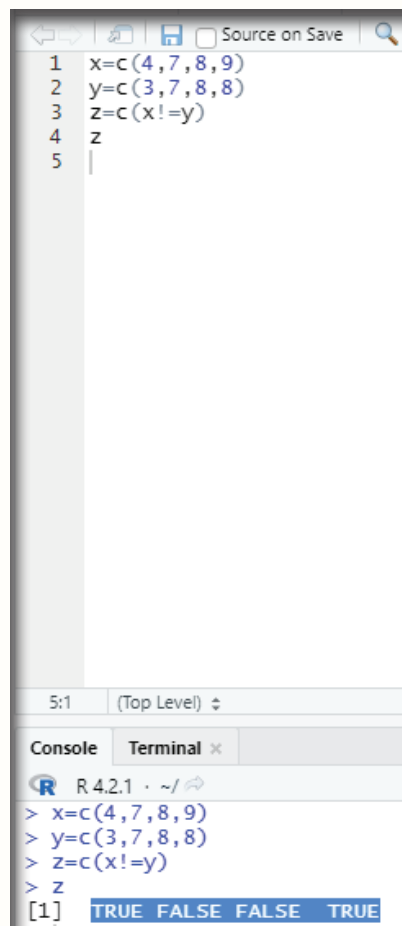
```
y=c(3,7,8,8)
```

```
z=c(x!=y)
```

```
z
```

Output:

TRUE FALSE FALSE TRUE



```
1 x=c(4,7,8,9)
2 y=c(3,7,8,8)
3 z=c(x!=y)
4 z
5 |

5:1 (Top Level)
Console Terminal x
R 4.2.1 ~ /
> x=c(4,7,8,9)
> y=c(3,7,8,8)
> z=c(x!=y)
> z
[1] TRUE FALSE FALSE TRUE
```

Image showing the use of `!=`

Logical operators:

Given below are the various logical operators supported in R language. It is applicable only to vectors of type logical, numeric or complex. All numbers greater than 1 is considered as logical value true.

Operator: &

This operator is called Element wise logical AND operator. It combines each element of the first vector with the corresponding element of the second vector and gives an output TRUE if both the elements are TRUE.

Code:

```
x = c(2,4,0, TRUE, 2+3i)
```

```
y = c(3,0,1, FALSE, 2+3i)
```

```
z = c(x&y)
```

```
z
```

Output:

```
TRUE FALSE FALSE FALSE TRUE
```

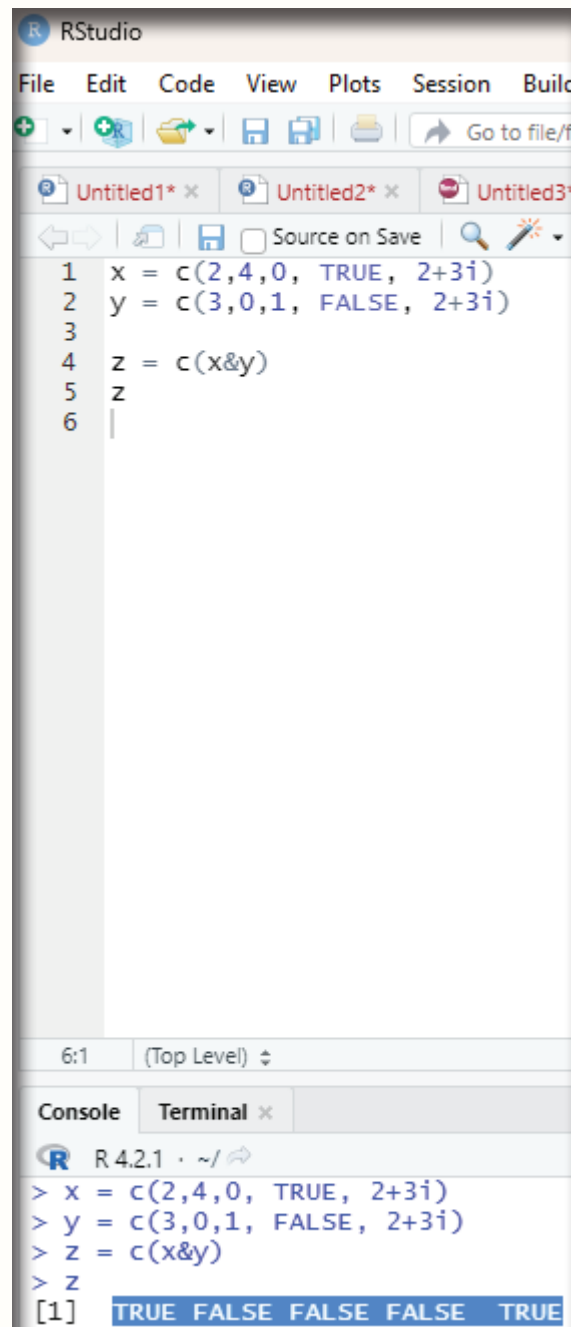
Considering the output the following explanation can be offered:

When the first value of both vectors are compared it can be seen both these values are more than 1 and hence both values are supposed to be TRUE. Since both values are TRUE the output generated shows the value TRUE.

When the second value of both vectors are compared it can be seen that the first vector has a value of more than one (hence should show TRUE, while the second value of the second vector is less than 1 and hence should display the value FALSE. Since both these values are not similar the output displays the value FALSE

Similarly the third value of both vectors displays dissimilar logical values hence they are reported in the output as FALSE.

The last value of the First and Second vectors are both more than 1 and hence the output displays the value TRUE.



The screenshot shows the RStudio interface. The source editor contains the following R code:

```
1 x = c(2,4,0, TRUE, 2+3i)
2 y = c(3,0,1, FALSE, 2+3i)
3
4 z = c(x&y)
5 z
6
```

The console at the bottom shows the execution of these commands:

```
> x = c(2,4,0, TRUE, 2+3i)
> y = c(3,0,1, FALSE, 2+3i)
> z = c(x&y)
> z
[1] TRUE FALSE FALSE FALSE TRUE
```

Image showing the use of & operator

Operator: |

This is also known as element wise logical OR operator. It combines each element of the first vector with the corresponding element of second vector and gives an output as TRUE if one of the elements is TRUE.

Code:

$x=c(3,5,7,TRUE)$

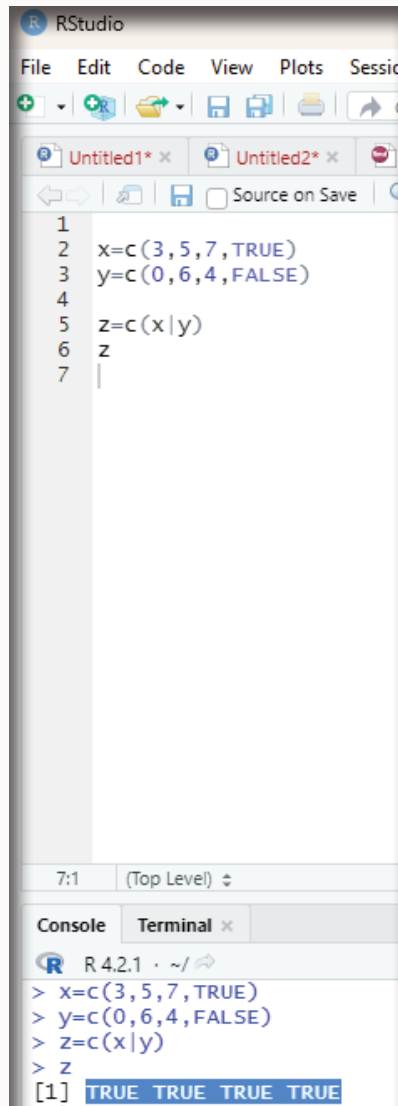
$y=c(0,6,4,FALSE)$

$z=c(x|y)$

z

Output:

TRUE TRUE TRUE TRUE



The screenshot shows the RStudio interface. The script editor contains the following code:

```
1  
2 x=c(3,5,7,TRUE)  
3 y=c(0,6,4,FALSE)  
4  
5 z=c(x|y)  
6 z  
7 |
```

The console at the bottom shows the execution of the code:

```
> x=c(3,5,7,TRUE)  
> y=c(0,6,4,FALSE)  
> z=c(x|y)  
> z  
[1] TRUE TRUE TRUE TRUE
```

Image showing the use of or operator

Operator: !

This operator is also known as logical NOT operator. This operator takes each element of the vector and gives the opposite logical value.

Code:

```
x=c(4,0,5,TRUE)
```

```
print (!x)
```

Output:

FALSE TRUE FALSE FALSE



```
1 x=c(4,0,5,TRUE)
2
3 print (!x)
4

4:1 (Top Level)
Console Terminal x
R 4.2.1 ~ /
> x=c(4,0,5,TRUE)
> print (!x)
[1] FALSE TRUE FALSE FALSE
```

Image showing the use of NOT operator

The logical operators && and || considers only the first element of the vectors and give a vector of single element as output.

Operator - &&

This operator is also known as Logical AND operator. It takes the first element of both the vectors and gives the TRUE only if both are true.

Code:

```
v <- c(3,0,TRUE,2+2i)
```

```
t <- c(1,3,TRUE,2+3i)
```

```
print(v&& t)
```

Output:

TRUE

Operator ||:

This is also known as Logical OR operator. This operator takes the first element of both the vectors and gives the TRUE if one of them is TRUE.

Code:

```
v <- c(0,0,TRUE,2+2i)
```

```
t <- c(0,3,TRUE,2+3i)
```

```
print(v||t)
```

Output:

FALSE

Some of the other mathematical functions are:

Square root - sqrt

Logarithm - log

Exponential - exp

Reader is encouraged to try out all these functions.

Create a vector “x” with a sequence of numbers between 1 and 4. These numbers should increment by 0.5.

Code:

```
x <- seq(1,4, by=0.5)
```

```
x
```

```
sqrt(x)
```

Output:

```
x <- seq(1,4, by=0.5)
```

```
> x
```

```
[1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0
```

```
> sqrt(x)
```

```
[1] 1.000000 1.224745 1.414214 1.581139 1.732051 1.870829 2.000000
```

```
>
```

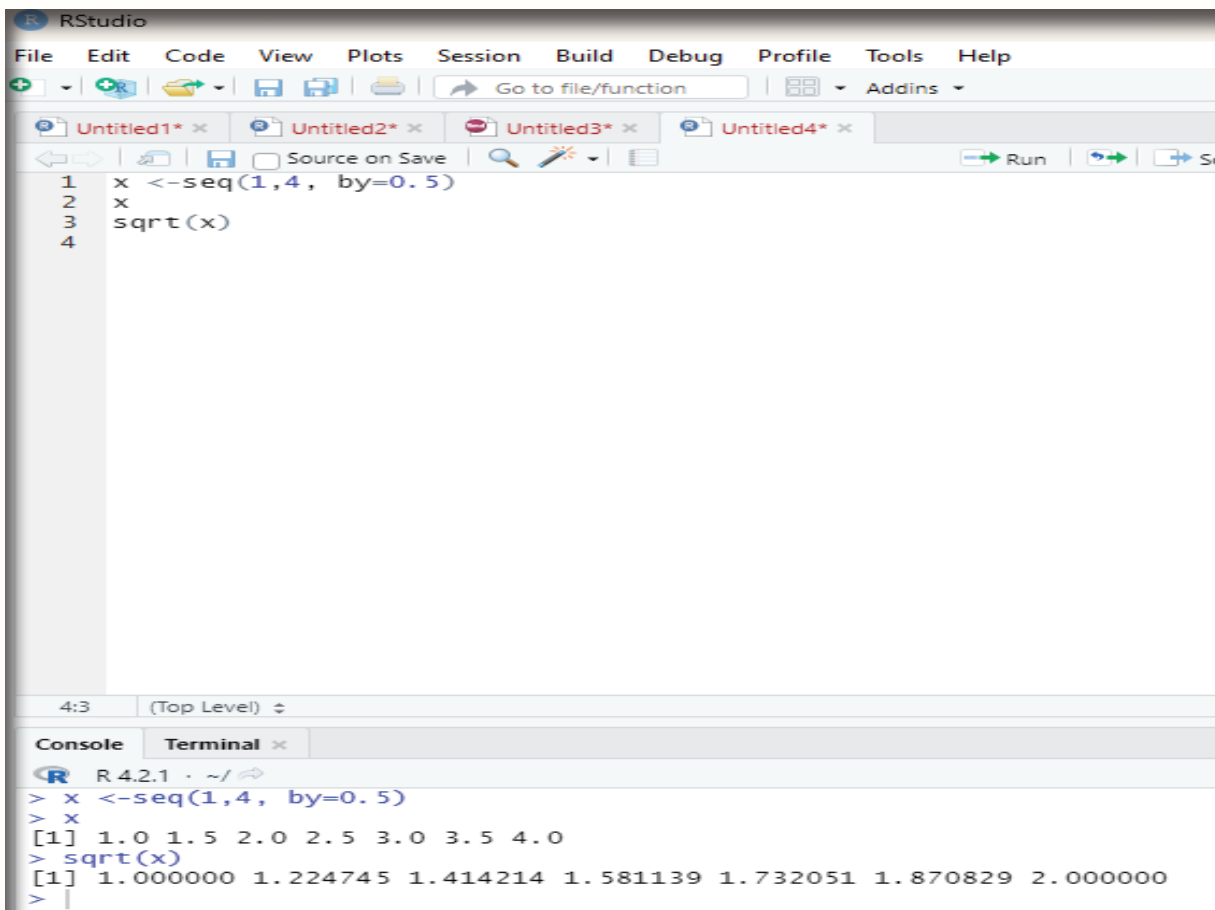


Image showing the above code in execution

Similarity Logarithmic value of x can be calculated using log command.

log(x)

Sine value can be calculated using sin command.

sin(x)

Output:

```
> log(x)
[1] 0.0000000 0.4054651 0.6931472 0.9162907 1.0986123 1.2527630 1.3862944
> sin(x)
[1] 0.8414710 0.9974950 0.9092974 0.5984721 0.1411200 -0.3507832
[7] -0.7568025
```

Assignment Operators:

These operators are used to assign values to vectors.

Left assignment:

```
<-
=
<<-
```

These operators can be used interchangeably.

v=c(1,2,3)

v<-c(1,2,3)

v<<-c(1,2,3)

c indicates concatenate in R language.

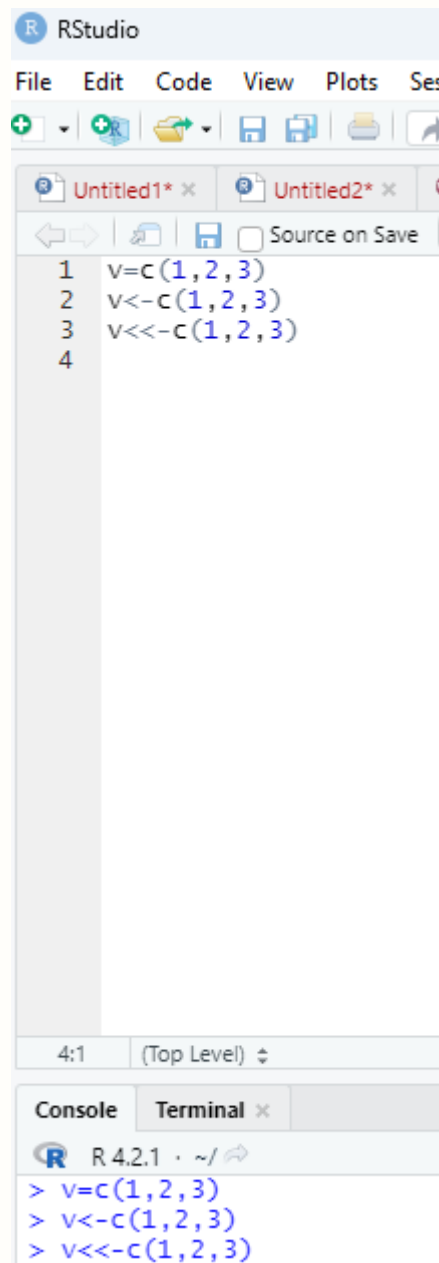


Image showing left assignment operators in use. They can be used interchangeably.

Right assignment operators:

->

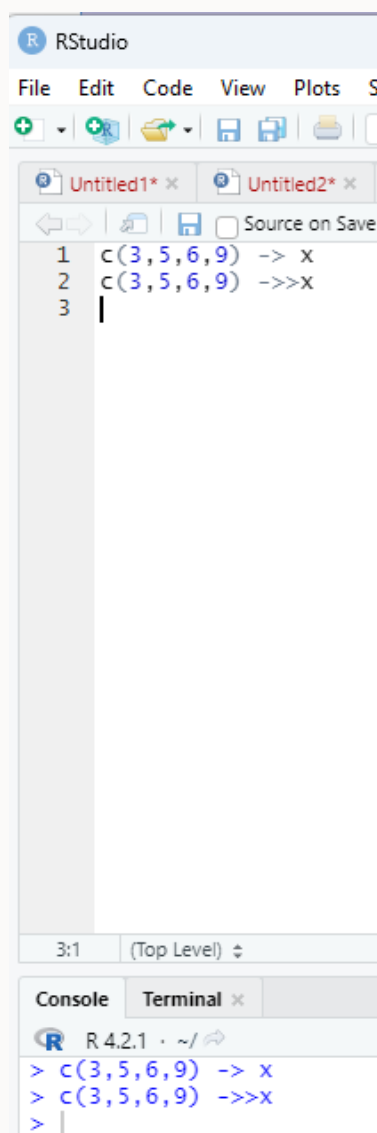
Example:

c(3,5,6,9) -> x

Note the code is reversed.

->>

c(3,5,6,9) ->>x



Miscellaneous operators:

: (colon operator): This operator creates the series of numbers in sequence for a vector.

```
x <- 2:8
```

```
print(x)
```

Output:

```
2 3 4 5 6 7 8
```



Image showing colon operator

`%in%` Operator:

This operator is used to identify if an element belongs to a vector.

Example:

Two vector are created.

```
x <- 8
```

```
y <- 12
```

Condition vector. Inside this vector the condition is entered, which is a series of numbers between 1 and 10 with an incremental value of 1 between them.

```
z <- 1:10
```

```
print (x%in%z)
```

This is to query whether variable x contains any value between 1 and 10.

```
print (y%in%z)
```

Output:

```
x <- 8
> y <- 12
> z <- 1:10
> print (x%in%z)
[1] TRUE
> print (y%in%z)
[1] FALSE
>
```

`%*%` Matrix multiplication:

This operator is used to multiply a matrix with its transpose.

Code:

```
M = matrix( c(2,6,5,1,10,4), nrow = 2,ncol = 3,byrow = TRUE)
```

```
t = M %*% t(M)
```

```
print(t)
```

Output:

```
  [,1] [,2]
[1,] 65  82
[2,] 82 117
```

Statistical summary function:

There are many inbuilt functions in R that helps the researcher in data analysis. These are rather simple to use.

Function	Purpose
Mean	Mean
Median	Median
sd	Standard deviation
var	variance
mad	Median Absolute deviation
min	Minimum
max	maximum
Range	Range of values (minimum and maximum)
sum	Total sum

The first argument to all these functions is the data and should be single vector of values.

Example:

```
age<-c(24,34,12,56,72,84)
```

```
median(age)
```

Output : 45

```
mad(age)
```

Output : 35.5824

```
range (age)
```

Output: 12 84

If missing data is there in the vector values then extra care needs to be taken while running these functions. When there are missing values in the vector values running these functions will give a return value of NA. This can be avoided by using the argument na.rm = (TRUE/FALSE).

Example:

```
age<-c(24,34,12,56,72,NA)
```

```
median (age, na.rm=TRUE)
```

Output - 34



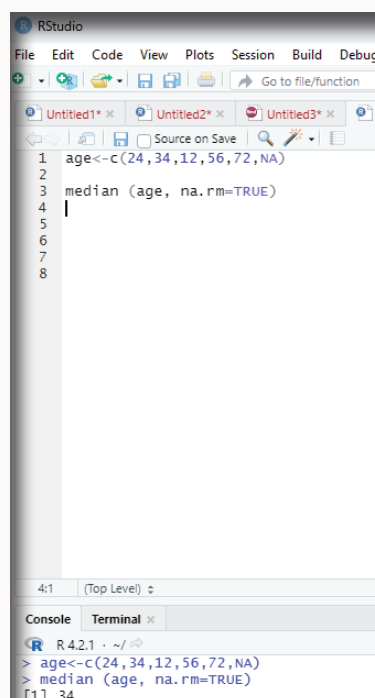
The screenshot shows the RStudio interface with a script editor and a console. The script editor contains the following code:

```
1 age<-c(24,34,12,56,72,84)
2
3 median(age)
4 mad(age)
5 range(age)
6
7
8
9
```

The console shows the output of the executed code:

```
R 4.2.1 ~ /
> age<-c(24,34,12,56,72,84)
> median(age)
[1] 45
> mad(age)
[1] 35.5824
> range(age)
[1] 12 84
```

Image showing the use of various statistical summary functions



The screenshot shows the RStudio interface with a script editor and a console. The script editor contains the following code:

```
1 age<-c(24,34,12,56,72,NA)
2
3 median (age, na.rm=TRUE)
4
5
6
7
8
```

The console shows the output of the executed code:

```
R 4.2.1 ~ /
> age<-c(24,34,12,56,72,NA)
> median (age, na.rm=TRUE)
[1] 34
```

Image showing how to handle missing data

Simulation and statistical distributions:

User who is working with statistical distributions in R, there are functions available for all of the common distributions and all common actions. All of these functions follow the same pattern of naming, which starts with a single letter to identify what the user wants to do and is followed by the R code name for the distribution.

R Code for Statistical Distribution

Distribution	R Code	Distribution	R Code
Normal	norm	Poisson	pois
Binominal	binom	Exponential	exp
Uniform	unif	Weibull	weibull
Beta	beta	Gamma	gamma
F	f	Chi-squared	chisq

The list shown above is not a complete one. More can be found in the help pages by searching for the name of the distribution. The user will have to combine the name of the distribution with a letter that determines whether to sample or calculate the quantiles.

Letter	Purpose	First Argument	Example
d	Probability density function	x (quantiles)	dnorm (1.64)
p	Cumulative probability density function	q (quantiles)	pnorm (1.64)
q	Quantile function	p (probabilities)	qnorm (0.95)
r	Random sampling	n (sample size)	rnorm (100)

Table showing various Distribution functions

Normal distribution has the arguments mean and sd that are set to the Standard Normal defaults (0 and 1) whereas the Poisson distribution has the argument lambda., which does not have a default value set. In general the arguments will be set to the “standard” values for the distribution. If the distribution does not have a standard, default values will not be set.

Example:

rnorm (5)

Output:

[1] 0.3321504 -0.1533315 -0.8361300 0.5362145 -1.6682728

rpois (5, lambda=3)

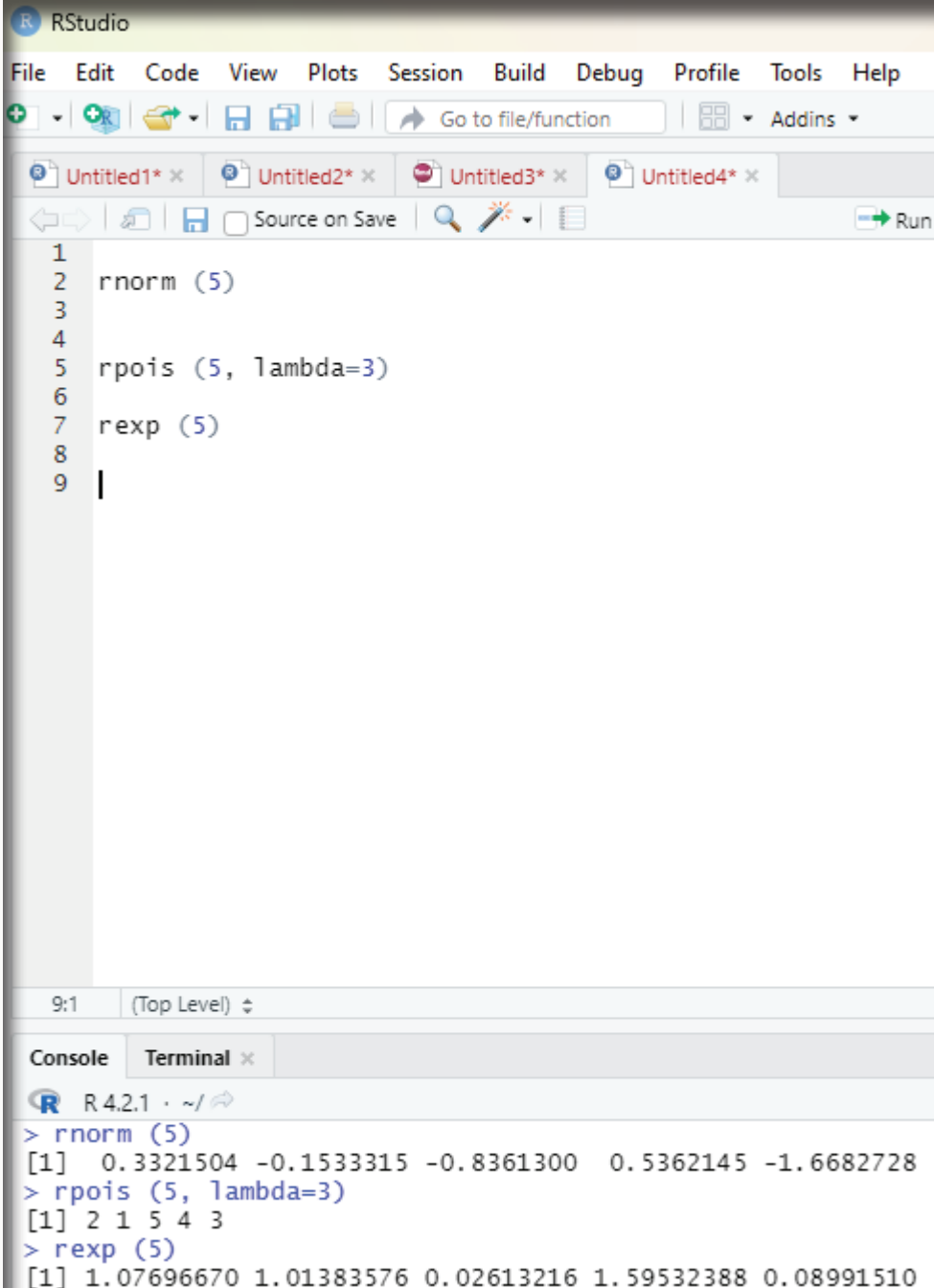
Output:

```
[1] 2 1 5 4 3
```

rexp (5)

Output:

```
[1] 1.07696670 1.01383576 0.02613216 1.59532388 0.08991510
```



The screenshot shows the RStudio interface. The source editor contains the following code:

```
1  
2 rnorm (5)  
3  
4  
5 rpois (5, lambda=3)  
6  
7 rexp (5)  
8  
9 |
```

The console at the bottom shows the output of these commands:

```
R 4.2.1 · ~/   
> rnorm (5)  
[1] 0.3321504 -0.1533315 -0.8361300 0.5362145 -1.6682728  
> rpois (5, lambda=3)  
[1] 2 1 5 4 3  
> rexp (5)  
[1] 1.07696670 1.01383576 0.02613216 1.59532388 0.08991510
```

Image showing codes for various types of distribution

The above codes allows the user to simulate values from a distribution. If the user needs to generate samples from the existing data then the function sample should be used. This function allows the user to specify the vector the sample is desired from, the number of samples needed by the user, whether the user wants to replace the values or not, and whether the user desires to change the probability of sampling particular value, which are equal by default.

As an example the function “sample” is applied to the vector of ages.

```
age = c(5,7,19,22,35,76,45,34)
sample (age, size =5)
```

Output:

```
[1] 45 35 34 76 5
```

Replace argument if used allows values to be sampled again when it is set to TRUE. If it is set to FALSE a value cannot be sampled again after it has been sampled once.

```
sample(age, size = 5, replace = TRUE)
```

Output:

```
[1] 76 76 5 22 45
```

Recreating simulated values in R Programming:

If the user desires to recreate the random samples from the samples one will need to set the random seed. This can be done using function set.seed. This takes an integer value to indicate the seed to use. This function can be used to change the type of random number generator used.

Example for generating random numbers from normal distribution:

Random numbers from a normal distribution can be generated using rnorm() function. The user will have to specify the number of samples to be generated. One can also specify the mean and standard deviation of the distribution. If these values are not provided the distribution defaults to 0 mean and 1 standard deviation.

Code to generate 1 random number

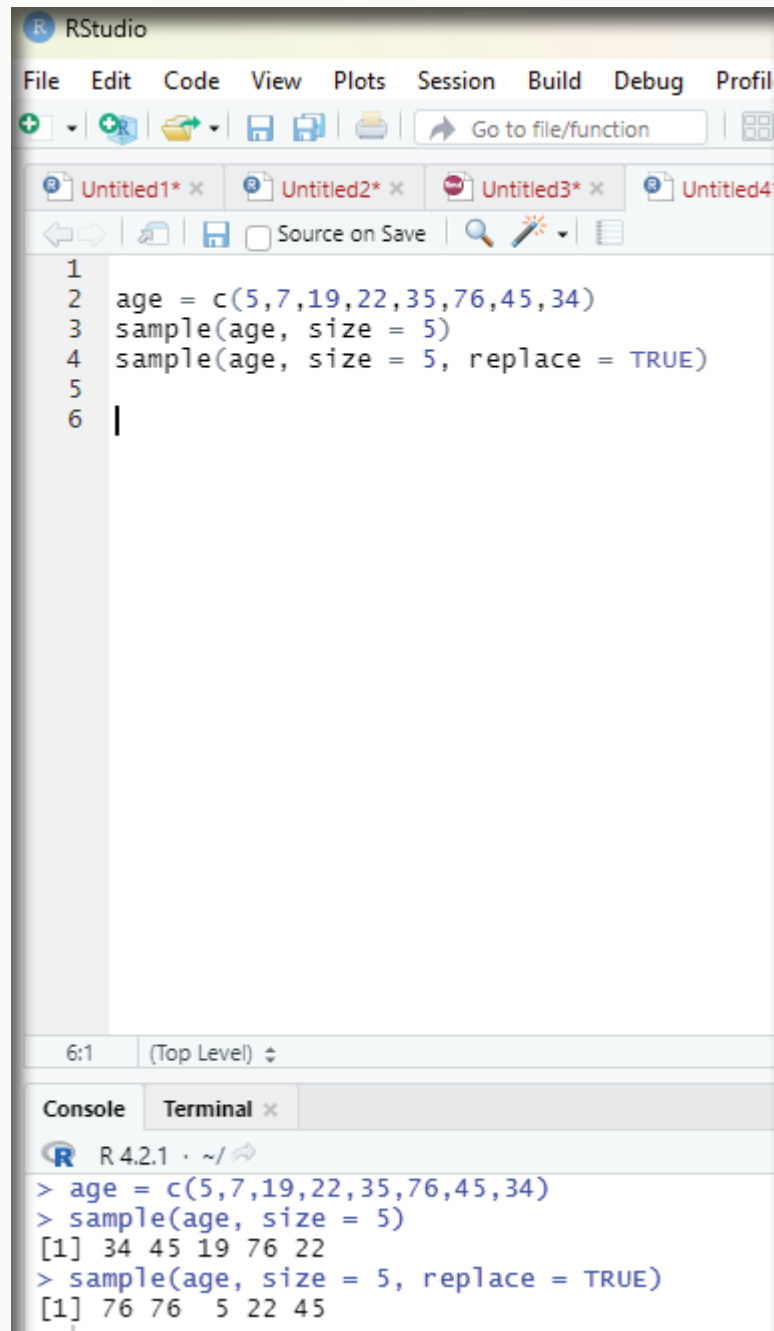
```
rnorm(1)
```

Output:

```
0.8418733
```

Code to generate 3 random numbers.

```
rnorm (3)
```



The image shows the RStudio interface. The source editor contains the following R code:

```
1  
2 age = c(5,7,19,22,35,76,45,34)  
3 sample(age, size = 5)  
4 sample(age, size = 5, replace = TRUE)  
5  
6 |
```

The console shows the execution of the code:

```
R 4.2.1 · ~/   
> age = c(5,7,19,22,35,76,45,34)  
> sample(age, size = 5)  
[1] 34 45 19 76 22  
> sample(age, size = 5, replace = TRUE)  
[1] 76 76 5 22 45
```

Image showing the use of sample function

Output:

0.6218214 -1.2239963 -1.5102920

Code to for providing the user's own mean and standard deviation.

```
rnorm (3, mean=10, sd=2)
```

Output:

9.487026 8.168494 11.471801

Search and Replace function:

These are two very useful functions for working with character data:

grep - This function allows the user to search elements of a vector for a particular pattern.

gsub - This function replaces a particular pattern with a given string (gsub).

Example:

```
colorStrings <-c ("green", "blue", "orange", "light green", "indigo blue", "navy blue")
```

Code to search for red in the above character string.

```
grep ("blue", colorStrings, value=TRUE)
```

Output:

"blue" "indigo blue" "navy blue"

Search and Replace function:

These are two very useful functions for working with character data:

grep - This function allows the user to search elements of a vector for a particular pattern.

gsub - This function replaces a particular pattern with a given string (gsub).

Example:

```
colorStrings <-c ("green", "blue", "orange", "light green", "indigo blue", "navy blue")
```

Code to search for red in the above character string.

```
grep ("blue", colorStrings, value=TRUE)
```

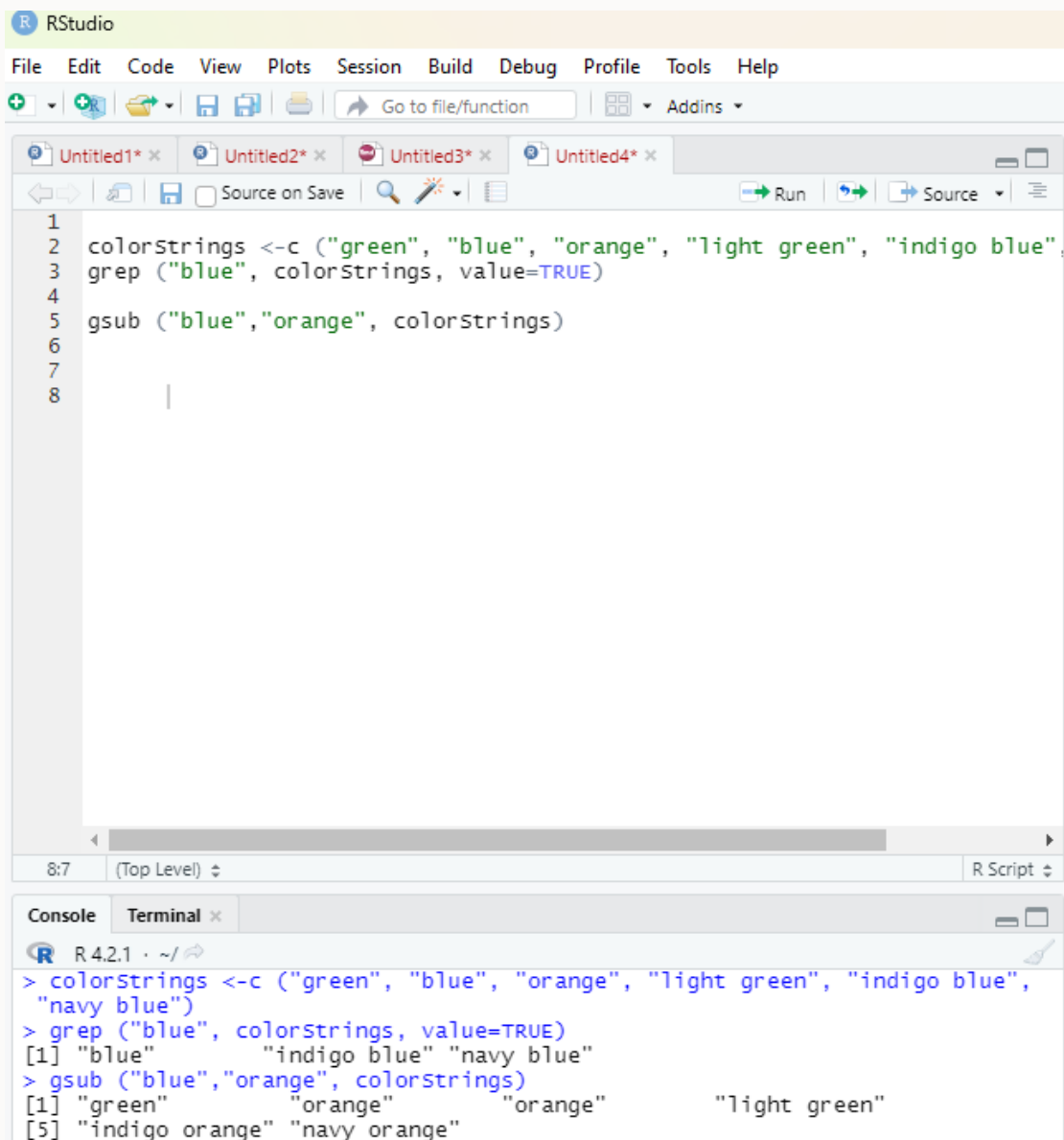
Output:

"blue" "indigo blue" "navy blue"

```
gsub ("blue", "orange", colorStrings)
```

Output:

```
[1] "green" "orange" "orange" "light green"  
[5] "indigo orange" "navy orange"
```



The screenshot shows the RStudio interface with a script editor and a console. The script editor contains the following code:

```
1  
2 colorStrings <-c ("green", "blue", "orange", "light green", "indigo blue",  
3 grep ("blue", colorStrings, value=TRUE)  
4  
5 gsub ("blue","orange", colorStrings)  
6  
7  
8
```

The console shows the execution of the code:

```
R 4.2.1 ~/  
> colorStrings <-c ("green", "blue", "orange", "light green", "indigo blue",  
"navy blue")  
> grep ("blue", colorStrings, value=TRUE)  
[1] "blue" "indigo blue" "navy blue"  
> gsub ("blue","orange", colorStrings)  
[1] "green" "orange" "orange" "light green"  
[5] "indigo orange" "navy orange"
```

Image showing the use of grep and gsub functions

Functions in R Programming

Functions in R allows the use to perform a number of tasks with a simple command. Writing functions is more or less similar with most programming languages. Creating own functions by the user is a powerful aspect of R. It allows the user to “wrap up” a series of steps into a simple container. In this way the user can capture common workflows and utilities and call them when needed instead of producing long, verbose scripts of repeated code snippets that can be difficult to manage. The function performs its task and returns control to the interpreter as well as any result which may be stored in other objects.

Components of Functions:

Function name - This is the name of the function. It is stored in R environment as an object with this name.

Arguments - An argument is a placeholder. When a function is invoked, a value is passed to the argument. Arguments are optional; that is a function may contain no arguments. Arguments also can have default values.

Function Body - The function body contains a collection of statements that defines what the function does.

Return value - The return value of a function is the last expression in the function body to be evaluated.

upper.tri function:

This function allows the user to identify values in the upper triangle of a matrix.

Syntax: upper.tri(x,diag)

x: Matrix object

diag: Boolean value to include diagonal

Code:

```
# R program to print the upper triangle of a matrix.
```

```
# Code to create a matrix.
```

```
mat <- matrix(c(1:9), 3,3, byrow=TRUE)
```

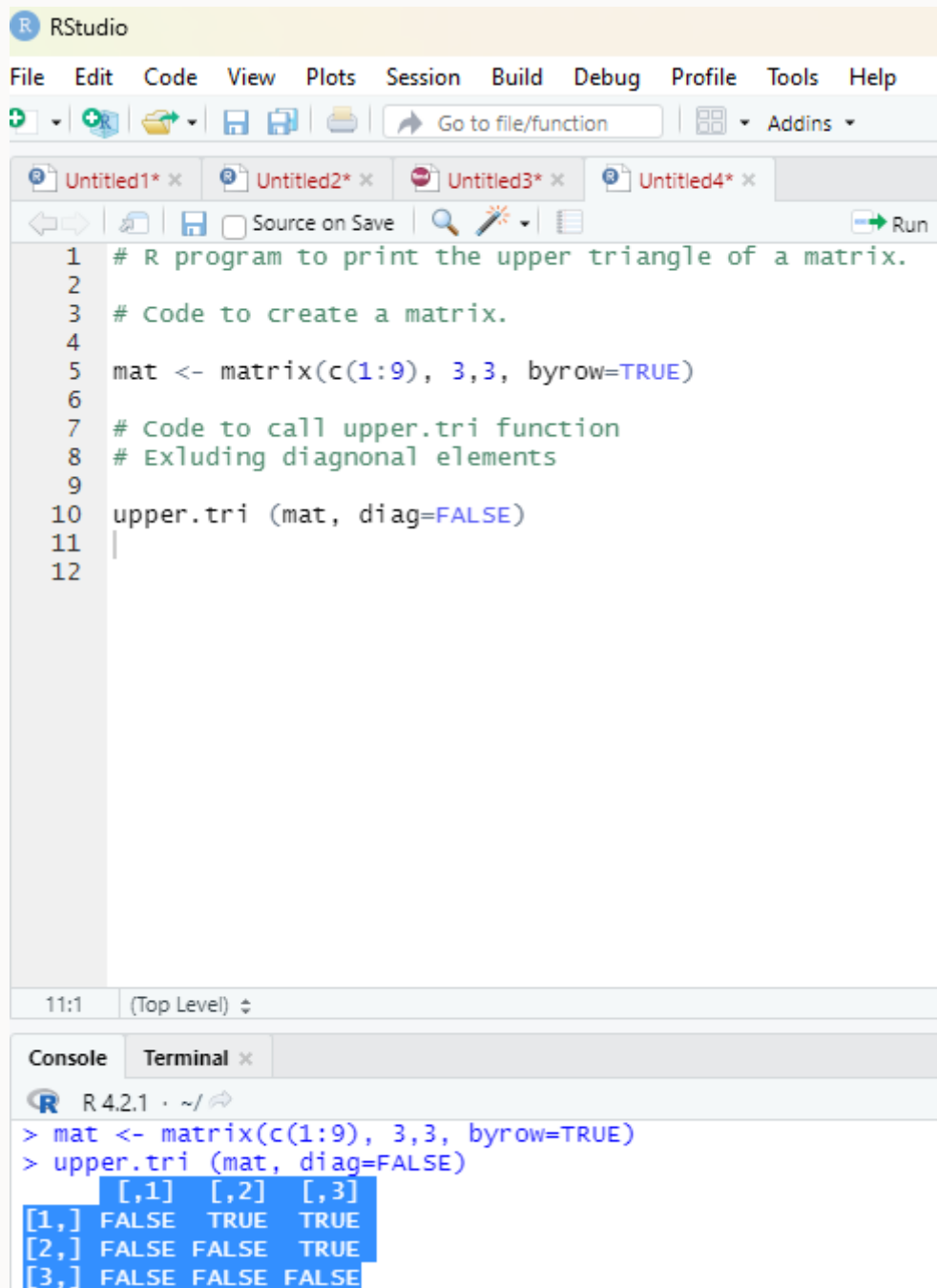
```
# Code to call upper.tri function
```

```
# Excluding diagonal elements
```

```
upper.tri(mat, diag=FALSE)
```

Output:

```
[,1] [,2] [,3]  
[1,] FALSE TRUE TRUE  
[2,] FALSE FALSE TRUE  
[3,] FALSE FALSE FALSE
```



The screenshot shows the RStudio interface. The source editor contains the following R code:

```
1 # R program to print the upper triangle of a matrix.  
2  
3 # Code to create a matrix.  
4  
5 mat <- matrix(c(1:9), 3,3, byrow=TRUE)  
6  
7 # Code to call upper.tri function  
8 # Excluding diagonal elements  
9  
10 upper.tri (mat, diag=FALSE)  
11  
12
```

The console output shows the execution of the code:

```
> mat <- matrix(c(1:9), 3,3, byrow=TRUE)  
> upper.tri (mat, diag=FALSE)  
      [,1] [,2] [,3]  
[1,] FALSE TRUE TRUE  
[2,] FALSE FALSE TRUE  
[3,] FALSE FALSE FALSE
```

Image showing upper.tri function in use

Output showing the contents of the Matrix:

```
[,1] [,2] [,3]
[1,]  1  2  3
[2,]  4  5  6
[3,]  7  8  9
```

Output seen after using upper.tri (mat, diag=FALSE) code.

```
[,1] [,2] [,3]
[1,] FALSE TRUE TRUE
[2,] FALSE FALSE TRUE
[3,] FALSE FALSE FALSE
```

Output seen after using upper.tri(mat, diag = TRUE)

```
[,1] [,2] [,3]
[1,] TRUE TRUE TRUE
[2,] FALSE TRUE TRUE
[3,] FALSE FALSE TRUE
```

In mathematics (linear algebra), a triangular matrix is a special kind of square matrix. A square matrix is called lower triangular if all the entries above the main diagonal are zero. Similarly, a square matrix is called upper triangular if all the entries below the main diagonal are zero.

A square matrix is said to be lower triangular matrix if all the elements above its main diagonal are zero.

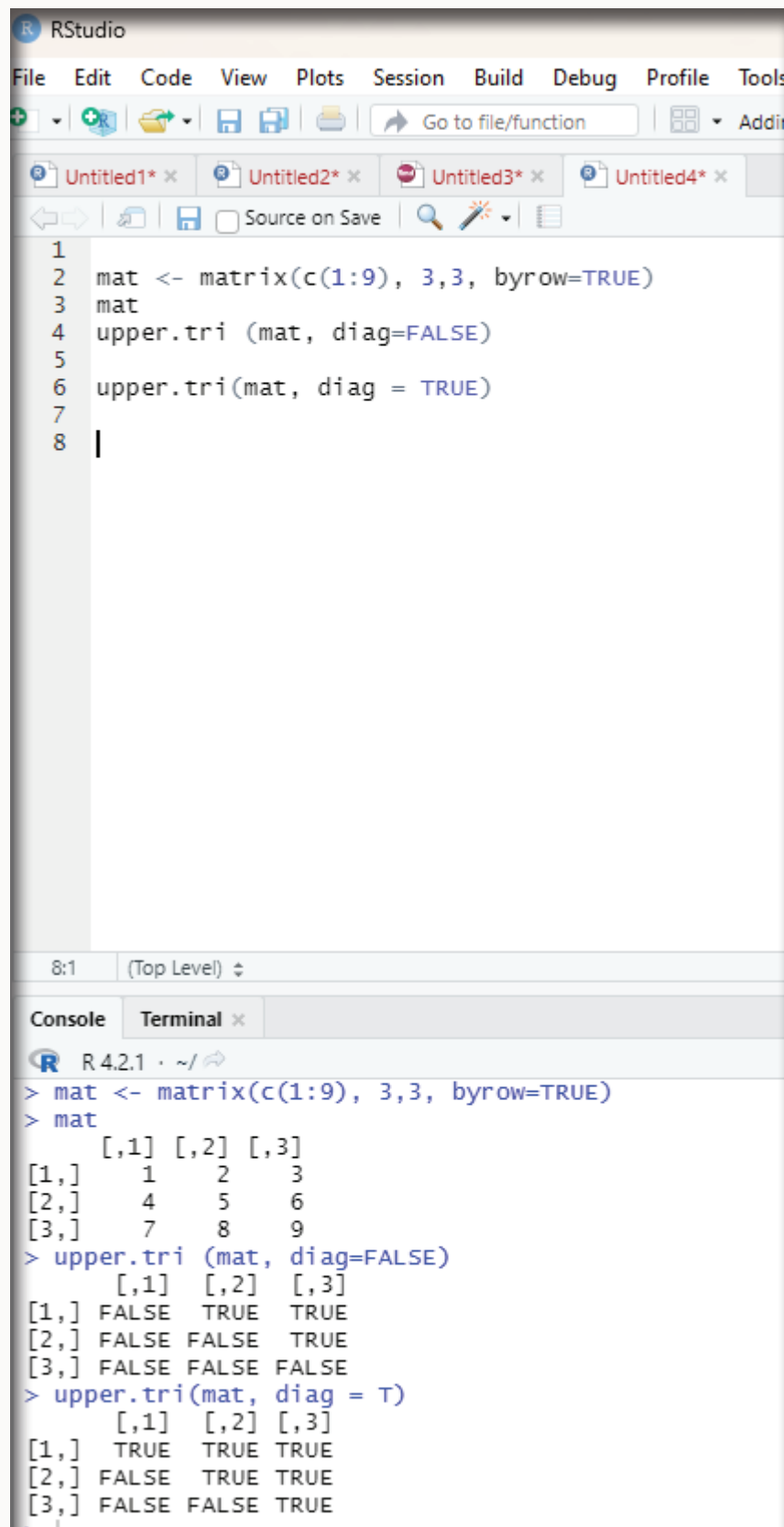
A square matrix is said to be an upper triangular matrix if all the elements below the main diagonal are zero.

$$B = \begin{bmatrix} 2 & 0 & 0 \\ 1 & 5 & 0 \\ 1 & 1 & 2 \end{bmatrix}$$

(Lower triangular matrix)

$$A = \begin{bmatrix} 2 & -1 & 3 \\ 0 & 5 & 2 \\ 0 & 0 & 2 \end{bmatrix}$$

(Upper triangular matrix)



The screenshot shows the RStudio interface. The script editor contains the following R code:

```
1  
2 mat <- matrix(c(1:9), 3,3, byrow=TRUE)  
3 mat  
4 upper.tri (mat, diag=FALSE)  
5  
6 upper.tri(mat, diag = TRUE)  
7  
8 |
```

The console shows the execution of the code:

```
R 4.2.1 ~/  
> mat <- matrix(c(1:9), 3,3, byrow=TRUE)  
> mat  
      [,1] [,2] [,3]  
[1,]    1    2    3  
[2,]    4    5    6  
[3,]    7    8    9  
> upper.tri (mat, diag=FALSE)  
      [,1] [,2] [,3]  
[1,] FALSE TRUE  TRUE  
[2,] FALSE FALSE TRUE  
[3,] FALSE FALSE FALSE  
> upper.tri(mat, diag = T)  
      [,1] [,2] [,3]  
[1,]  TRUE  TRUE  TRUE  
[2,] FALSE  TRUE  TRUE  
[3,] FALSE FALSE  TRUE
```

Image showing the use of upper.tri function with diag (true and false) arguments

Functions typically contains more than one line of code. The script window is preferred to the console window while developing functions.

Naming a function:

A function is an R object and hence can be named like any other R object. The name can be:

Of any length.

Contain any combinations of letters, numbers, underscores and period characters.

Cannot start with a number.

Creating a simple function:

The user can create a simple function in R using the function keyword. Curly brackets are used to contain the body of the function.

Example:

```
addOne <- function (x) {x+1}
```

This function adds 1 to any input object.

```
addOne (x=2.5)
```

Output:

3.5

Types of functions in R language:

Built in function:

R has many in-built functions which can be directly called in the program without defining them first. One can also create and use customized functions referred to as user defined functions. Some of the in-built functions available in R are:

```
seq()  
mean()  
max()  
sum(x)  
paste(...)
```

Examples:

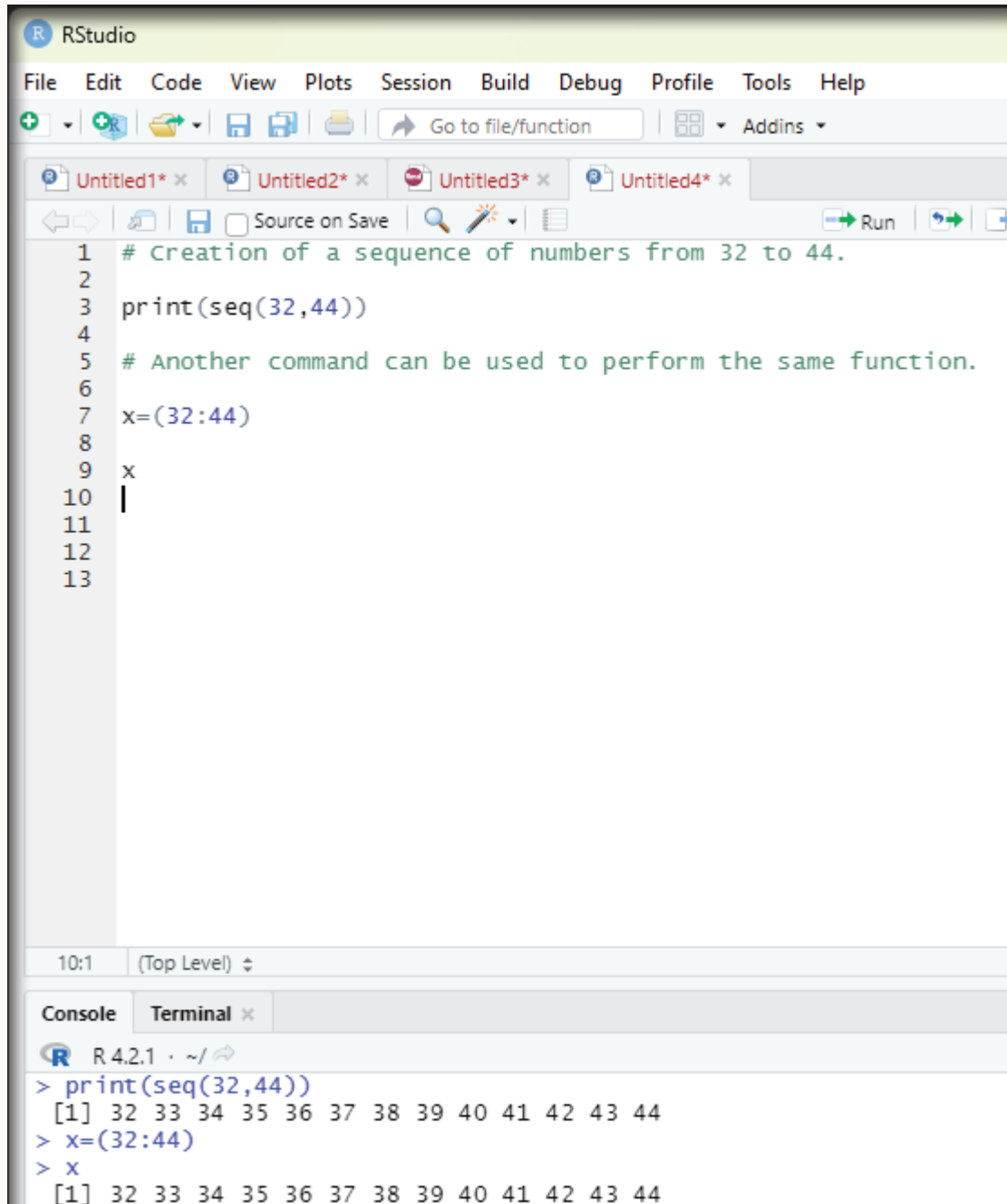
1. Creation of a sequence of numbers from 32 to 44.

```
print(seq(32,44))
```

Another command can be used to perform the same function using :

$x = (32:44)$

x



The screenshot shows the RStudio interface with four untitled files open. The active file, 'Untitled4*', contains the following R code:

```
1 # Creation of a sequence of numbers from 32 to 44.
2
3 print(seq(32,44))
4
5 # Another command can be used to perform the same function.
6
7 x=(32:44)
8
9 x
10
11
12
13
```

The console at the bottom shows the output of the commands:

```
R 4.2.1 ~ /
> print(seq(32,44))
[1] 32 33 34 35 36 37 38 39 40 41 42 43 44
> x=(32:44)
> x
[1] 32 33 34 35 36 37 38 39 40 41 42 43 44
```

Image showing creation of sequence of numbers

2. Finding mean of numbers from 25 to 82.

```
print(mean(25:82))
```

Output generated : 53.5

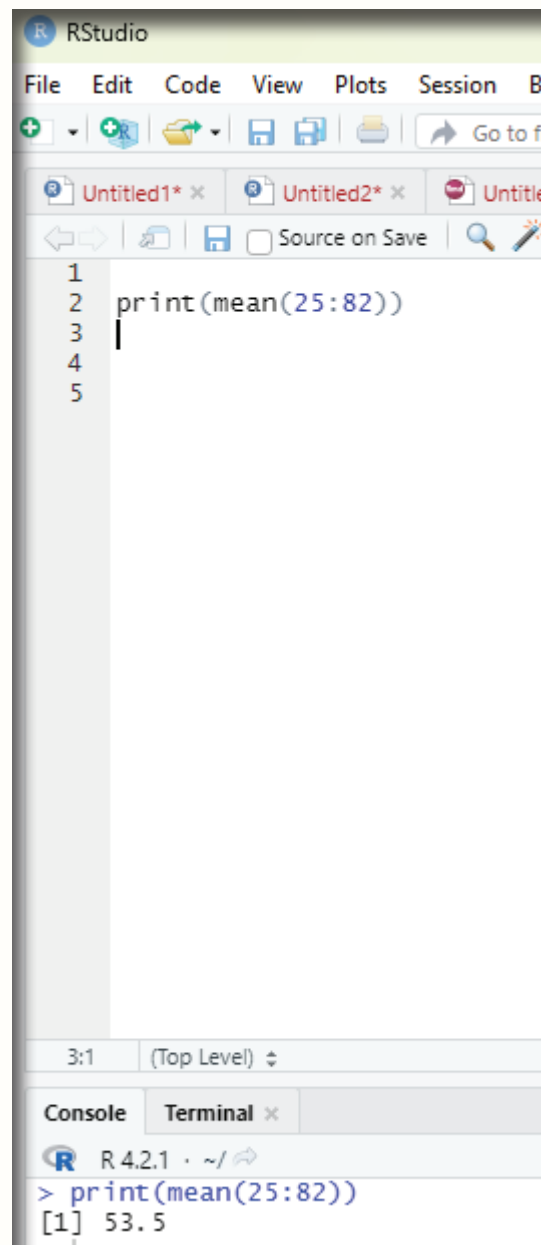
A screenshot of the RStudio interface. The top menu bar includes File, Edit, Code, View, Plots, Session, and Bu. Below the menu is a toolbar with icons for file operations and a 'Go to file' button. The file explorer shows three open files: 'Untitled1*', 'Untitled2*', and 'Untitled3'. The source editor shows a script with five lines: line 1 is empty, line 2 contains 'print(mean(25:82))', and lines 3, 4, and 5 are empty. The console at the bottom shows the command '> print(mean(25:82))' and the output '[1] 53.5'. The status bar at the bottom indicates 'R 4.2.1 · ~/ ·'.

Image showing calculation of mean value of a series of numbers

3. Finding sum of numbers from 41 to 68.

```
print(sum(41:68))
```

Output : 1526

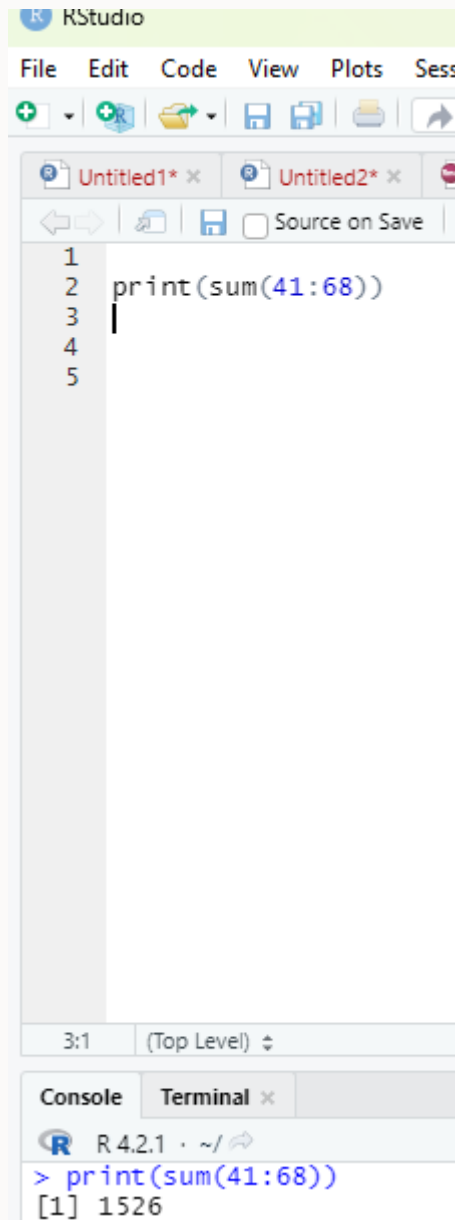


Image showing the sum of a series of numbers calculated

4. Finding the maximum from a series of values

```
x=c(12, 15, 3, 22, 18,43)  
print(max(x))
```

4. Finding the maximum from a series of values

```
x=c(12, 15, 3, 22, 18,43)  
print(max(x))
```

Output: 43

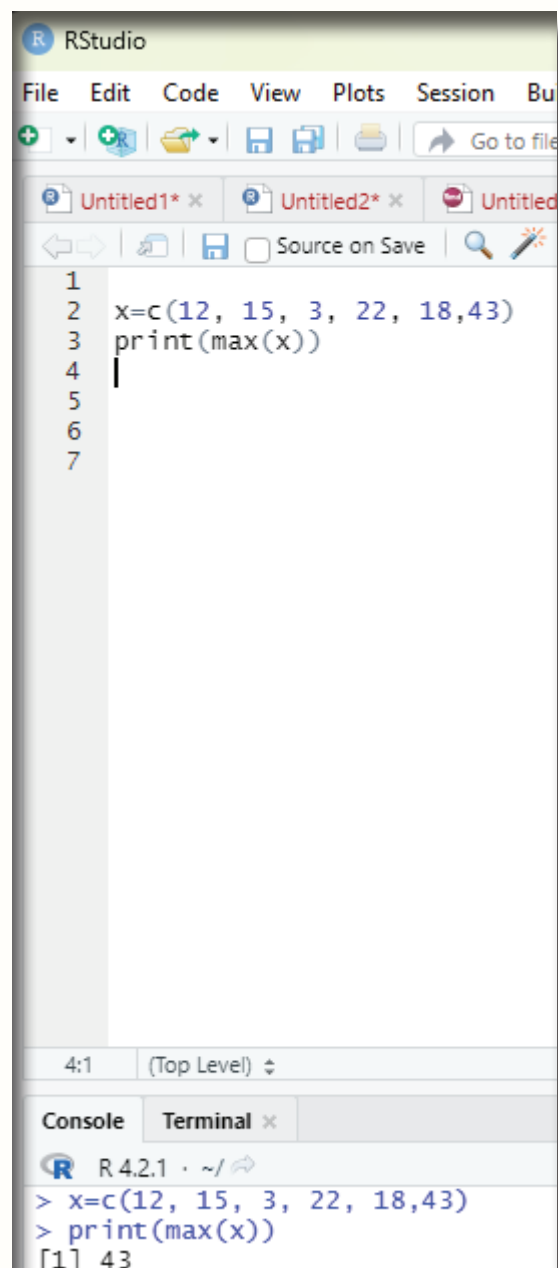


Image showing identifying the maximum value of a series of numbers

Example of user defined function:

1. The aim of this function is to check whether the value assigned to the variable x is even or odd.

Assign a value for the variable x.

x=22

Function code.

evenOdd = function(x) {if (x %% 2 == 0)

return("even")

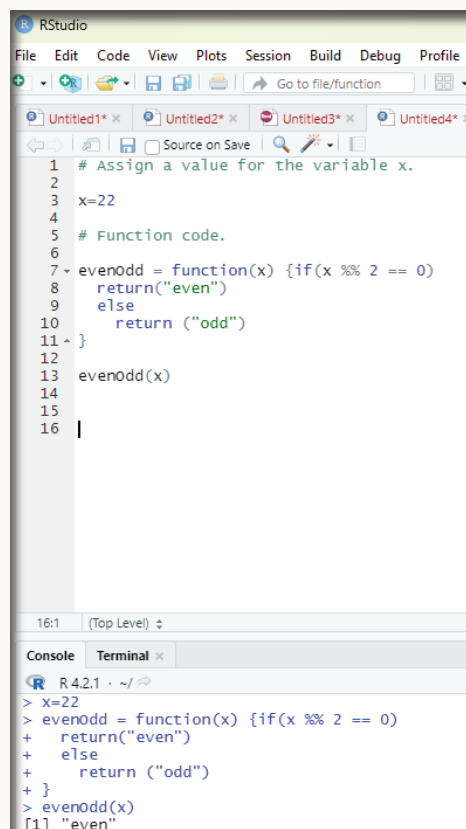
else

return ("odd")

}

print (evenOdd(x))

Output: "even"



The screenshot shows the RStudio interface. The script editor contains the following code:

```
1 # Assign a value for the variable x.  
2  
3 x=22  
4  
5 # Function code.  
6  
7 evenOdd = function(x) {if(x %% 2 == 0)  
8   return("even")  
9   else  
10    return ("odd")  
11 }  
12  
13 evenOdd(x)  
14  
15  
16 |
```

The console shows the execution of the code:

```
> x=22  
> evenOdd = function(x) {if(x %% 2 == 0)  
+   return("even")  
+   else  
+     return ("odd")  
+ }  
> evenOdd(x)  
[1] "even"
```

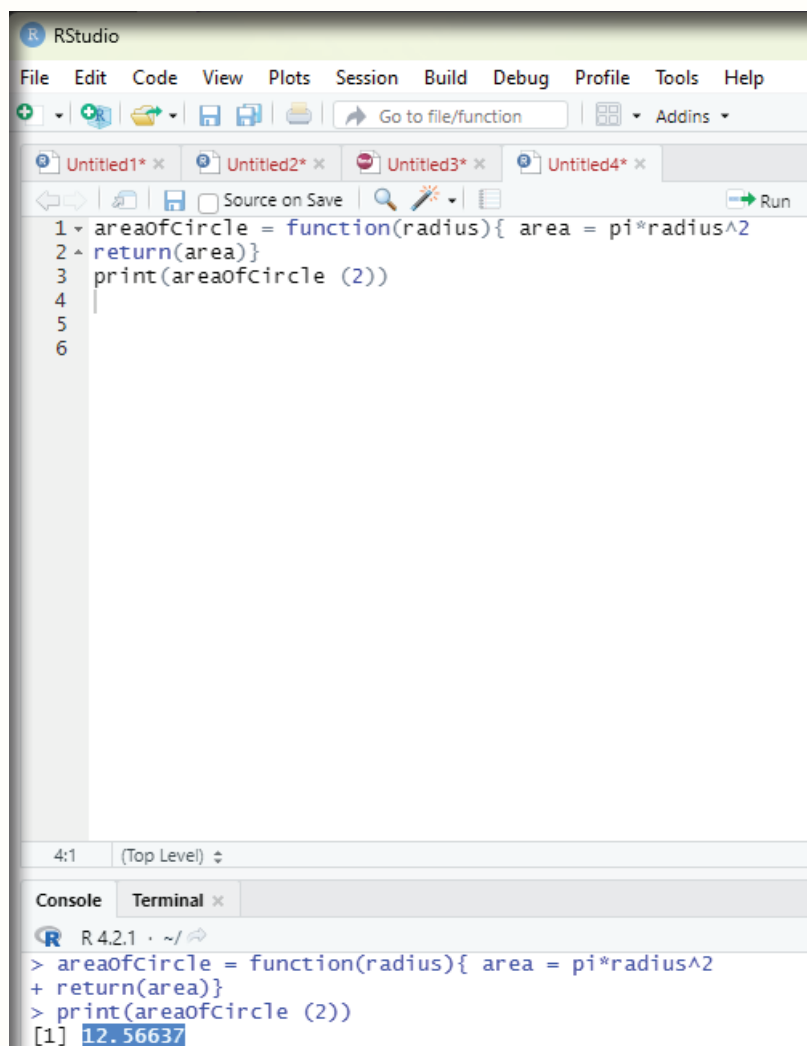
Image showing code that identifies odd and even numbers

2. The aim is to create a function in R that will take a single input and gives a single output. This function code should calculate the area of a circle when the radius is fed. The name of the function that needs to be created is 'areaOfCircle', and the arguments that are needed to be passed are the "radius" of the circle.

Code:

```
areaOfCircle = function(radius){ area = pi*radius^2  
  return(area)}  
print(areaOfCircle (2))
```

Outout: 12.56637



The screenshot shows the RStudio interface. The top menu bar includes File, Edit, Code, View, Plots, Session, Build, Debug, Profile, Tools, and Help. Below the menu bar is a toolbar with icons for file operations and a 'Go to file/function' search bar. The main editor window displays four untitled files. The first file, 'Untitled1*', contains the following R code:

```
1 areaOfCircle = function(radius){ area = pi*radius^2  
2 return(area)}  
3 print(areaOfCircle (2))  
4  
5  
6
```

The bottom panel shows the 'Console' tab. It displays the R prompt and the execution of the code:

```
> areaOfCircle = function(radius){ area = pi*radius^2  
+ return(area)}  
> print(areaOfCircle (2))  
[1] 12.56637
```

Image showing area of circle calculated

3. Creating a function to print squares of numbers in sequence:

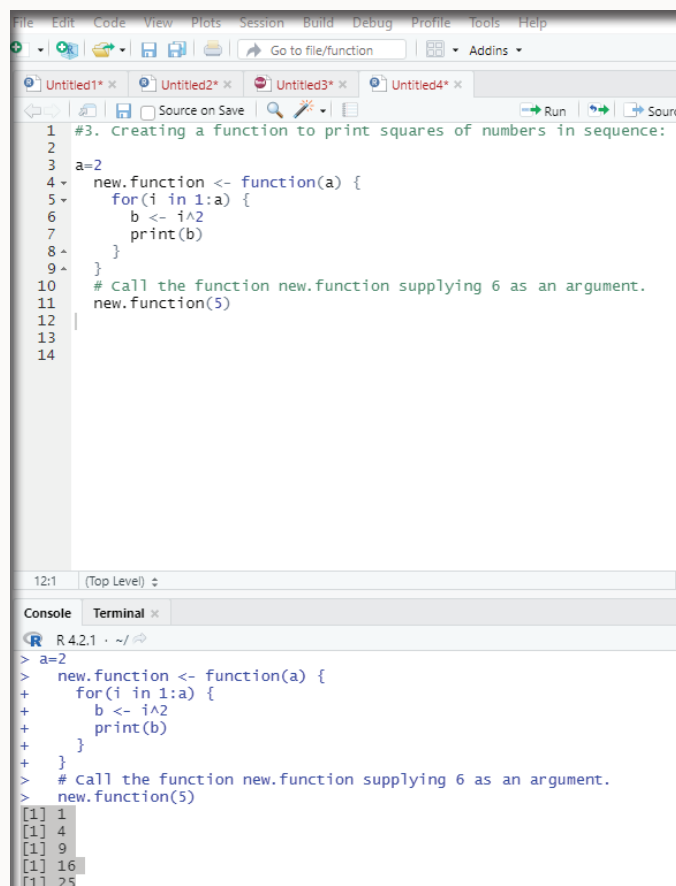
```
new.function <- function(a) {  
  for(i in 1:a) {  
    b <- i^2  
    print(b)  
  }  
}
```

Call the function new.function supplying 6 as an argument.

```
new.function(5)
```

Output:

```
[1] 1  
[1] 4  
[1] 9  
[1] 16  
[1] 25
```



The screenshot shows an R Studio interface with four untitled files. The active file contains the following R code:

```
1 #3. Creating a function to print squares of numbers in sequence:  
2  
3 a=2  
4 new.function <- function(a) {  
5   for(i in 1:a) {  
6     b <- i^2  
7     print(b)  
8   }  
9 }  
10  
11 # Call the function new.function supplying 6 as an argument.  
12 new.function(5)  
13  
14
```

The console window at the bottom shows the execution of the code, resulting in the following output:

```
> a=2  
> new.function <- function(a) {  
+   for(i in 1:a) {  
+     b <- i^2  
+     print(b)  
+   }  
+ }  
> # Call the function new.function supplying 6 as an argument.  
> new.function(5)  
[1] 1  
[1] 4  
[1] 9  
[1] 16  
[1] 25
```

Image showing the calculation of squares of numbers

4. Calling a function with argument values (by position and by name).

The arguments to a function call can be supplied in the same sequence as defined in the function, or they can be supplied in a different sequence but assigned to the names of the arguments.

Creating a function with arguments.

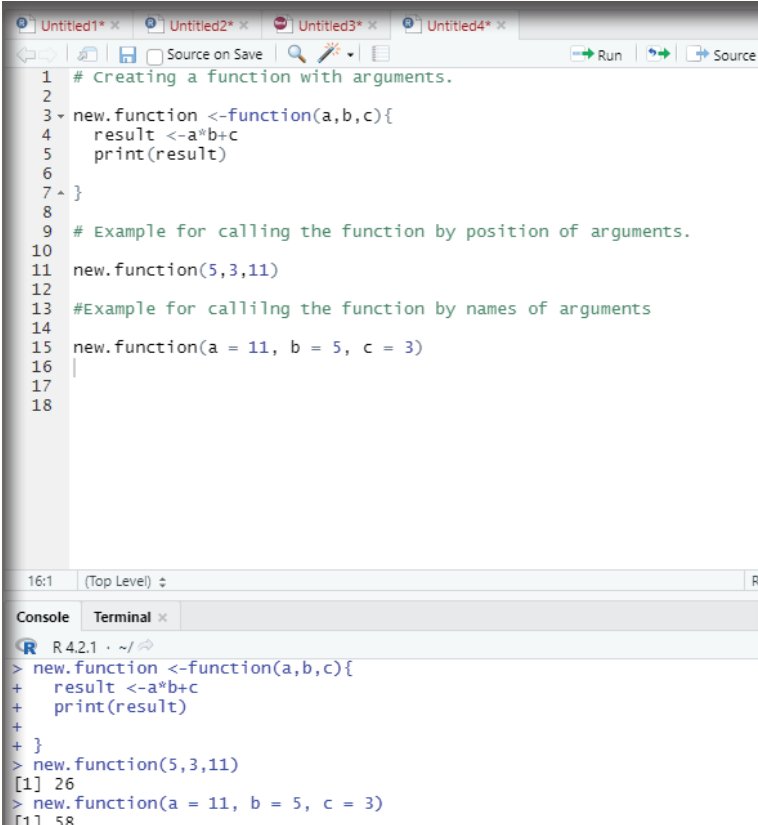
```
new.function <-function(a,b,c){  
  result <-a*b+c  
  print(result)
```

}# Example for calling the function by position of arguments.

```
new.function(5,3,11)
```

#Example for calling the function by names of arguments

```
new.function(a = 11, b = 5, c = 3)
```



The screenshot shows an RStudio interface with four untitled files. The active file contains the following R code:

```
1 # Creating a function with arguments.  
2  
3 new.function <-function(a,b,c){  
4   result <-a*b+c  
5   print(result)  
6  
7 }  
8  
9 # Example for calling the function by position of arguments.  
10  
11 new.function(5,3,11)  
12  
13 #Example for calling the function by names of arguments  
14  
15 new.function(a = 11, b = 5, c = 3)  
16  
17  
18
```

The console window at the bottom shows the execution of the code:

```
R 4.2.1 ~/  
> new.function <-function(a,b,c){  
+   result <-a*b+c  
+   print(result)  
+ }  
> new.function(5,3,11)  
[1] 26  
> new.function(a = 11, b = 5, c = 3)  
[1] 58
```

Image showing function with argument values by position and name

5. Lazy Evaluation of Function:

Arguments to functions are evaluated lazily. This means that they are evaluated only when needed by the function body.

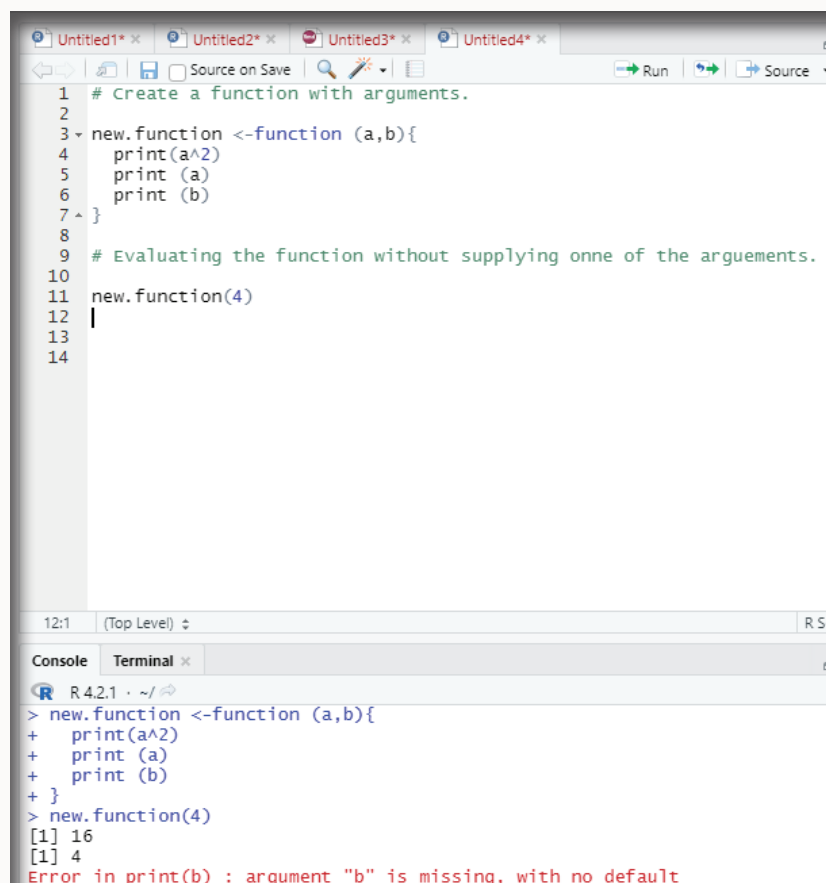
Create a function with arguments.

```
new.function <-function (a,b){  
  print(a^2)  
  print (a)  
  print (b)  
}
```

Evaluating the function without supplying one of the arguments.

```
new.function(4)
```

This will actually throw an error in printing b stating that argument “b” is missing.



```
1 # Create a function with arguments.  
2  
3 new.function <-function (a,b){  
4   print(a^2)  
5   print (a)  
6   print (b)  
7 }  
8  
9 # Evaluating the function without supplying one of the arguments.  
10  
11 new.function(4)  
12  
13  
14
```

```
R 4.2.1 ~/  
> new.function <-function (a,b){  
+   print(a^2)  
+   print (a)  
+   print (b)  
+ }  
> new.function(4)  
[1] 16  
[1] 4  
Error in print(b) : argument "b" is missing, with no default
```

Image showing lazy evaluation function

Number of arguments in a function:

By default, a function must be called with the correct number of arguments. If the function expects 2 arguments, one will have to call the function with two arguments, not more, and not less.

Example:

```
my_function <- function(fname, lname) {  
  paste(fname, lname)  
}my_function("Sam", "Peter")
```

Return values:

In order to make a function return a result the return() function should be used.

Example for the use of return() function:

```
multiplication_function <-function(x) {  
  return (5*x)  
}  
print (multiplication_function(2))  
print (multiplication_function(4))  
print (multiplication_function(5))
```


Nested functions:

There are two ways of creating Nested function.

1. Call a function within another function
2. Write a function within a function

Example - To call a function within another function:

```
Nested_function <- function(x, y) {  
  a <- x + y  
  return(a)  
}  
  
Nested_function(Nested_function(2,2), Nested_function(3,3))
```



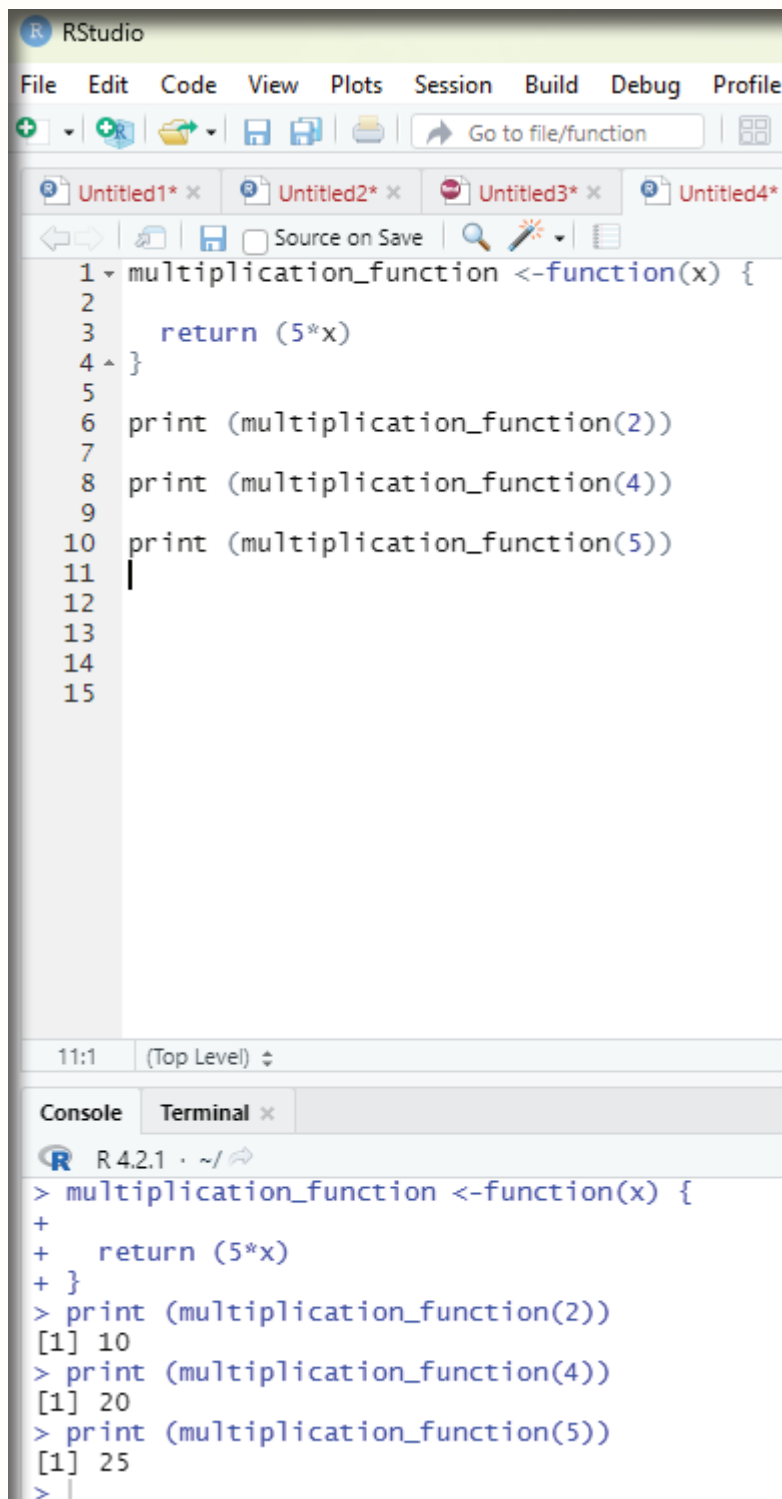
The image shows the RStudio interface. The top menu bar includes File, Edit, Code, View, Plots, Session, Build, Debug, and Profile. Below the menu is a toolbar with icons for file operations and a search bar labeled 'Go to file/function'. The editor pane shows four untitled files. The first file contains the following R code:

```
1 my_function <- function(fname, lname) {  
2   paste(fname, lname)  
3 }  
4  
5 my_function("Sam", "Peter")  
6  
7  
8
```

The status bar at the bottom of the editor shows '6:1 (Top Level)'. Below the editor is a console pane with the following output:

```
R 4.2.1 ~/  
> my_function <- function(fname, lname) {  
+   paste(fname, lname)  
+ }  
> my_function("Sam", "Peter")  
[1] "Sam Peter"
```

Image showing a function with two arguments



The image shows the RStudio interface. The top menu bar includes File, Edit, Code, View, Plots, Session, Build, Debug, and Profile. Below the menu is a toolbar with icons for file operations and a search bar labeled 'Go to file/function'. The editor pane shows four untitled files. The first file contains the following R code:

```
1 multiplication_function <-function(x) {  
2  
3   return (5*x)  
4 }  
5  
6 print (multiplication_function(2))  
7  
8 print (multiplication_function(4))  
9  
10 print (multiplication_function(5))  
11  
12  
13  
14  
15
```

The status bar at the bottom of the editor shows '11:1 (Top Level)'. Below the editor is a console pane with tabs for 'Console' and 'Terminal'. The console shows the execution of the code:

```
R 4.2.1 · ~/>  
> multiplication_function <-function(x) {  
+  
+   return (5*x)  
+ }  
> print (multiplication_function(2))  
[1] 10  
> print (multiplication_function(4))  
[1] 20  
> print (multiplication_function(5))  
[1] 25  
>
```

Image showing multiplication function

The screenshot displays the RStudio interface. The top menu bar includes File, Edit, Code, View, Plots, Session, Build, Debug, Profile, Tools, and Help. Below the menu is a toolbar with icons for file operations and a search bar. The editor pane shows four untitled files. The active file contains the following R code:

```
1 Nested_function <- function(x, y) {  
2   a <- x + y  
3   return(a)  
4 }  
5  
6 Nested_function(Nested_function(2,2), Nested_function(3,3))  
7  
8  
9  
10  
11  
12  
13
```

The status bar at the bottom indicates the current position is 11:1 at the top level. Below the editor is a console pane with the following output:

```
> Nested_function <- function(x, y) {  
+   a <- x + y  
+   return(a)  
+ }  
> Nested_function(Nested_function(2,2), Nested_function(3,3))  
[1] 10
```

Image showing Nested function

Explanation:

The function instructs x to add y.

The first Nested_function(2,2) is “x” of the main function.

The input Nested_function(3,3) is “y” of the main function.

The output hence is $(2+2) + (3+3) = 10$

Recursion:

R accepts function recursion, which means a defined function can call itself. This is a common mathematical and programming concept. It means that a function calls itself. This has the benefit of meaning that one can loop through data to reach a result.

The user should be careful with recursion function as it could easily slip into writing a function which never terminates thereby using excess amounts of memory or process power. Written correctly, it can be an efficient and mathematically elegant programming practice.

Example:

```
recursion <- function(k) {  
  if (k>0) {  
    result <- k+recursion(k-1)  
    print(result)  
  } else {  
    result = 0  
    return(result)  
  }  
}  
recursion(6)
```

R Global variables in functions:

Variables that are created outside of a function are known as global variables.

Example of creating a variable of a function and using it inside the function:

```
txt <- “very good”  
  
new_function <-function() {  
  paste(“R is”, txt)  
}  
new_function()
```

If the user tries to print txt, it will return the global variable which happens to be “very good”.

txt # print txt

The image shows the RStudio interface with a script editor and a console. The script editor contains a recursive function named `recursion` that takes an argument `k`. The function uses an `if` statement to check if `k` is greater than 0. If true, it calculates `result <- k + recursion(k-1)` and prints the result. If false, it sets `result = 0` and returns it. The function is then called with `recursion(6)`. The console shows the execution of the function, with the output of each recursive call printed on a new line.

```
1 recursion <- function(k) {  
2   if (k>0) {  
3     result <- k+recursion(k-1)  
4     print(result)  
5   } else {  
6     result = 0  
7     return(result)  
8   }  
9 }  
10  
11 }  
12  
13 recursion(6)  
14  
15  
16  
17  
18  
19  
20
```

14:1 (Top Level) ↕

Console Terminal ×

R 4.2.1 · ~/

```
+ if (k>0) {  
+   result <- k+recursion(k-1)  
+   print(result)  
+ } else {  
+   result = 0  
+   return(result)  
+ }  
+ }  
> recursion(6)  
[1] 1  
[1] 3  
[1] 6  
[1] 10  
[1] 15  
[1] 21
```

Image showing code for regression

Global assignment operator:

Normally, when one wants to create a variable inside a function, that variable is local and can only be used inside that function. To create a global variable inside a function, one can use the global assignment operator <<-

```
new_function <-function() {  
  txt <<- "very good"  
  paste("R is", txt)  
}  
new_function()  
  
print(txt)
```

Repeat rep() function:

code:

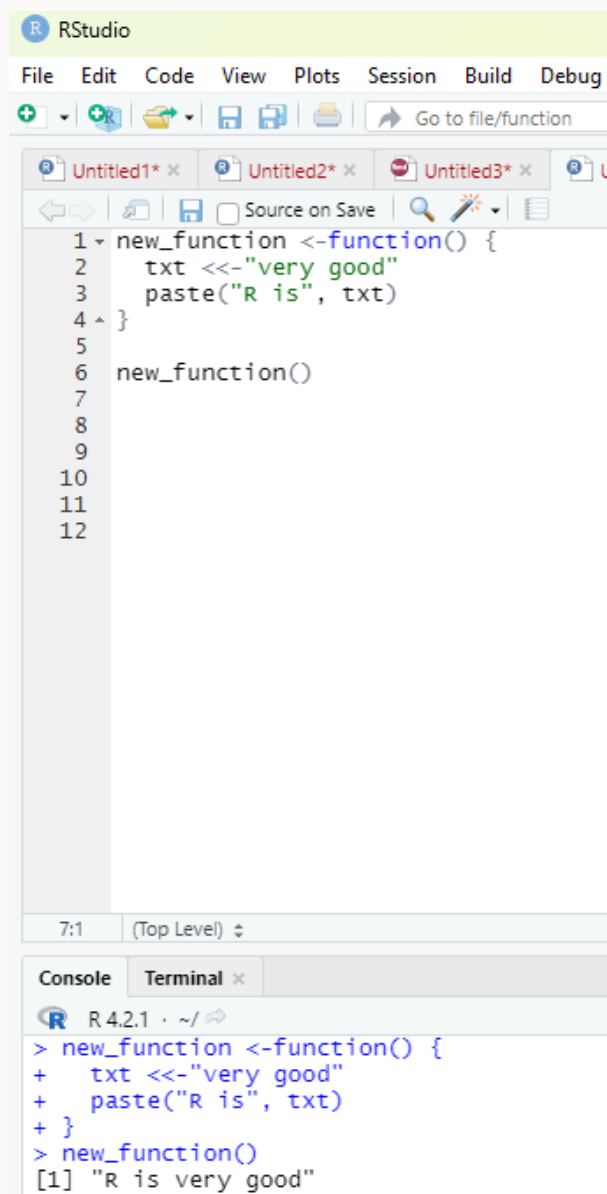
```
repeat_eachnumber <-rep(c(1,2,3,4), each =4)  
  
repeat_eachnumber
```

Repeat the sequence of the vector:

```
repeat_times <-rep(c(1,3,4,5), times =4)  
  
repeat_times
```

Repeating each value independently:

```
repeat_independent <-rep(c(1,3,5), times = c(1,5,8))  
repeat_independent
```



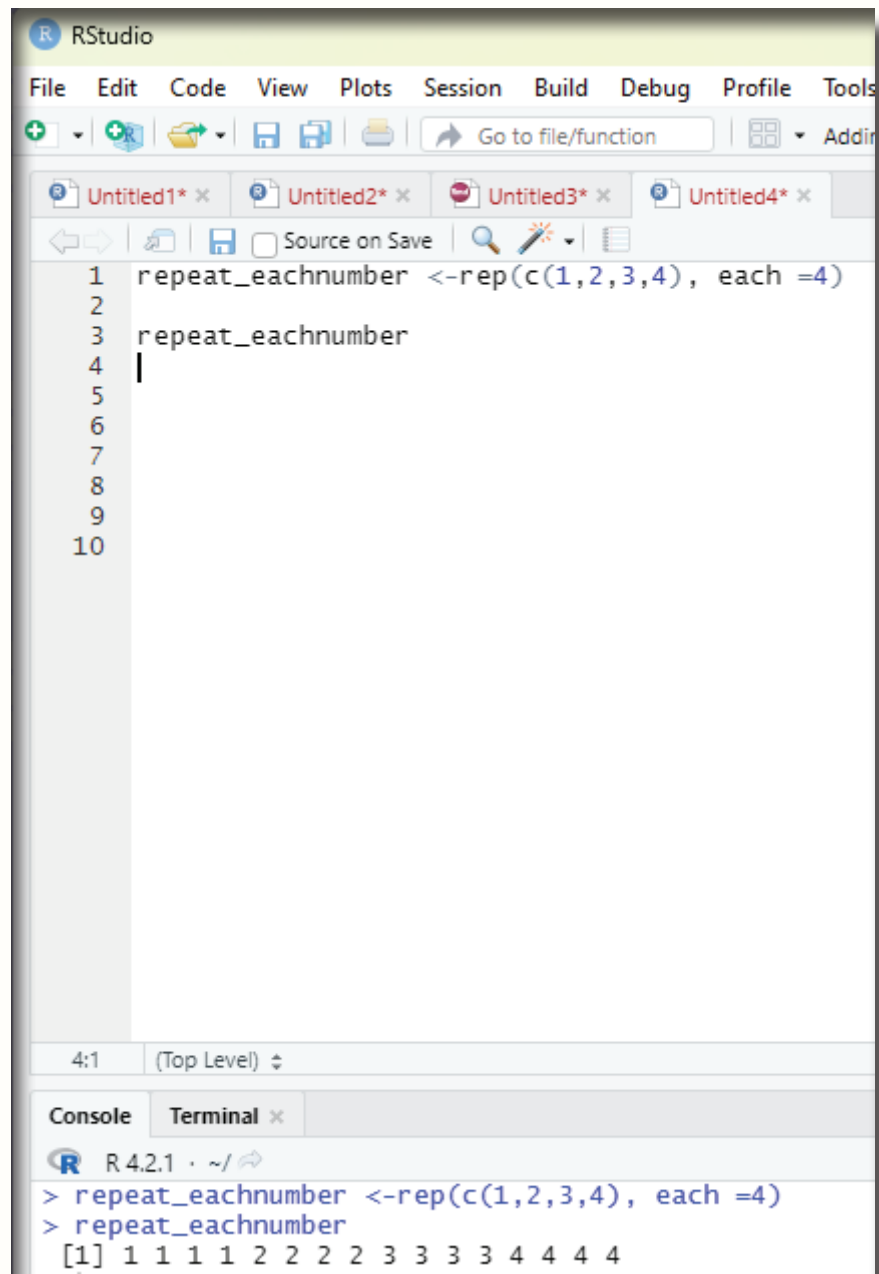
The image shows the RStudio interface. The top menu bar includes File, Edit, Code, View, Plots, Session, Build, and Debug. Below the menu is a toolbar with icons for file operations and a search bar labeled 'Go to file/function'. The main editor window displays three tabs: 'Untitled1*', 'Untitled2*', and 'Untitled3*'. The 'Untitled1*' tab is active, showing a function definition in R code:

```
1 new_function <-function() {  
2   txt <-"very good"  
3   paste("R is", txt)  
4 }  
5  
6 new_function()  
7  
8  
9  
10  
11  
12
```

Below the editor is a status bar showing '7:1' and '(Top Level)'. At the bottom is a console window with tabs for 'Console' and 'Terminal'. The 'Console' tab is active, showing the execution of the function:

```
> new_function <-function() {  
+   txt <-"very good"  
+   paste("R is", txt)  
+ }  
> new_function()  
[1] "R is very good"
```

Image showing global assignment operator function



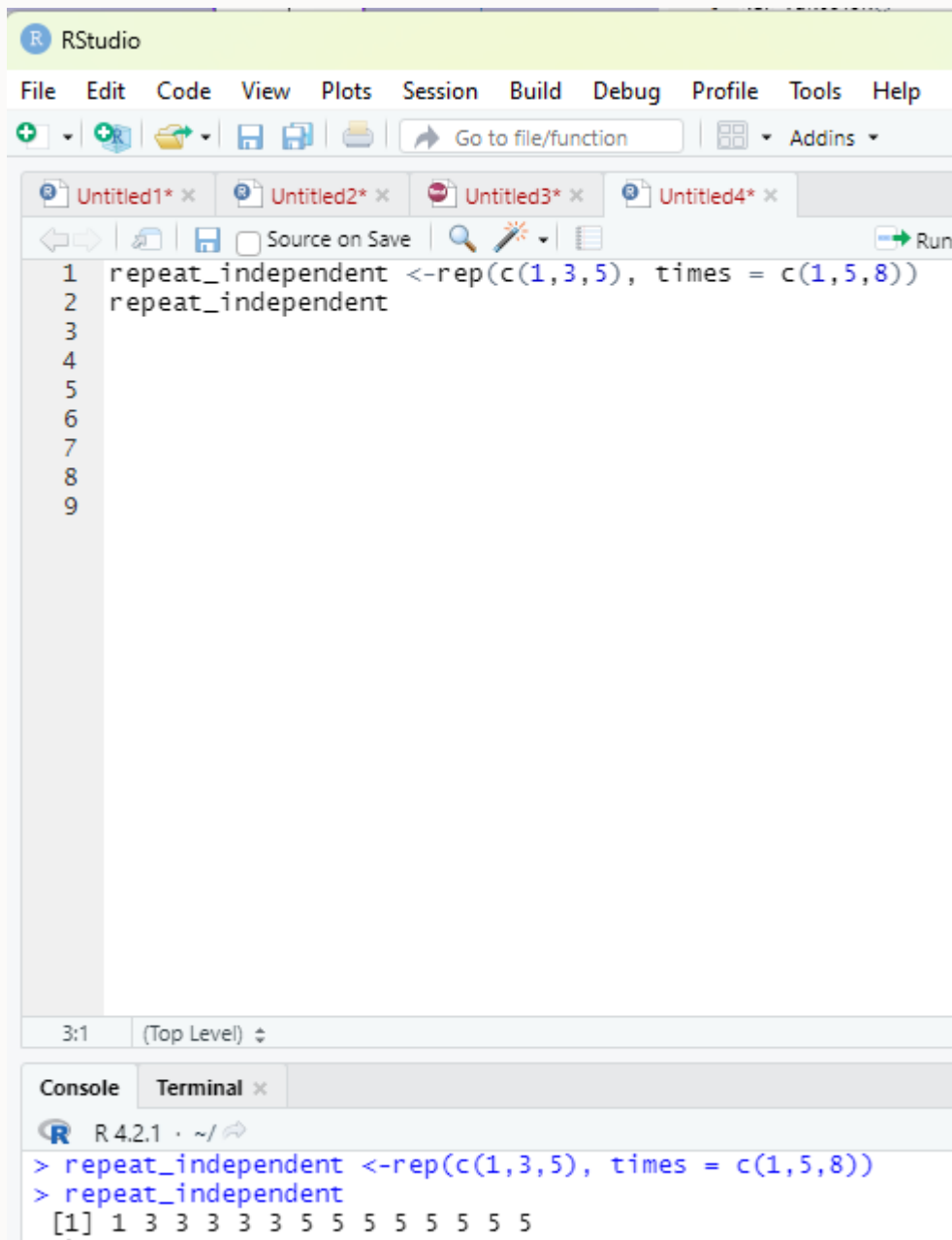
The screenshot shows the RStudio interface. The top menu bar includes File, Edit, Code, View, Plots, Session, Build, Debug, Profile, and Tools. Below the menu bar is a toolbar with icons for file operations and a search bar labeled 'Go to file/function'. The editor pane shows four untitled files. The first file, 'Untitled1*', contains the following R code:

```
1 repeat_eachnumber <-rep(c(1,2,3,4), each =4)
2
3 repeat_eachnumber
4
5
6
7
8
9
10
```

The console pane at the bottom shows the execution of the code:

```
R 4.2.1 ~/  
> repeat_eachnumber <-rep(c(1,2,3,4), each =4)  
> repeat_eachnumber  
[1] 1 1 1 1 2 2 2 2 3 3 3 3 4 4 4 4
```

Image showing repeat function



The screenshot shows the RStudio interface. The top menu bar includes File, Edit, Code, View, Plots, Session, Build, Debug, Profile, Tools, and Help. Below the menu is a toolbar with icons for adding files, saving, and running code. The main editor window displays four untitled files. The first file, 'Untitled1*', contains the following R code:

```
1 repeat_independent <-rep(c(1,3,5), times = c(1,5,8))
2 repeat_independent
3
4
5
6
7
8
9
```

The status bar at the bottom of the editor shows '3:1 (Top Level)'. Below the editor is a console window with the following output:

```
R 4.2.1 ~/> repeat_independent <-rep(c(1,3,5), times = c(1,5,8))
> repeat_independent
[1] 1 3 3 3 3 3 5 5 5 5 5 5 5 5
```

Image showing each value being repeated independently

Generating sequenced vectors:

Vectors can be created with numerical values in a sequence using : operator.

Example:

```
numbers <- 1:10
```

```
numbers
```

In order to create stepwise increment / decrement to a sequence of numbers in a vector the function seq() can be used. This function has three parameters. :from is where the sequence starts, to where the sequence stops, and by is the interval of the sequence.

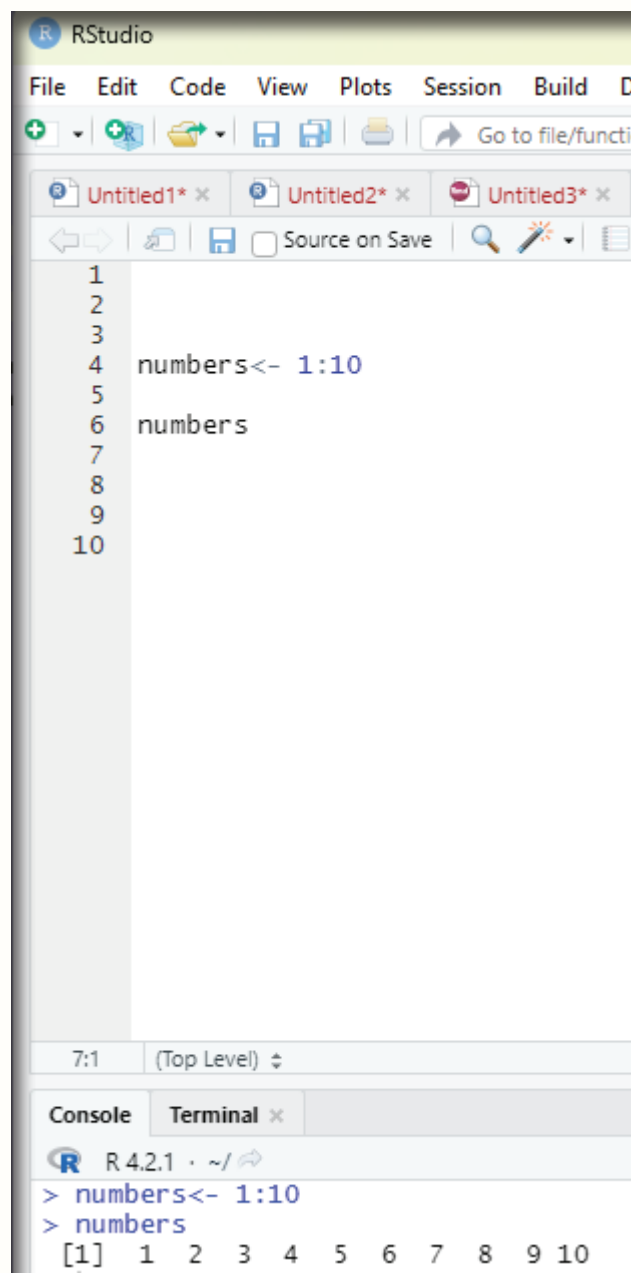
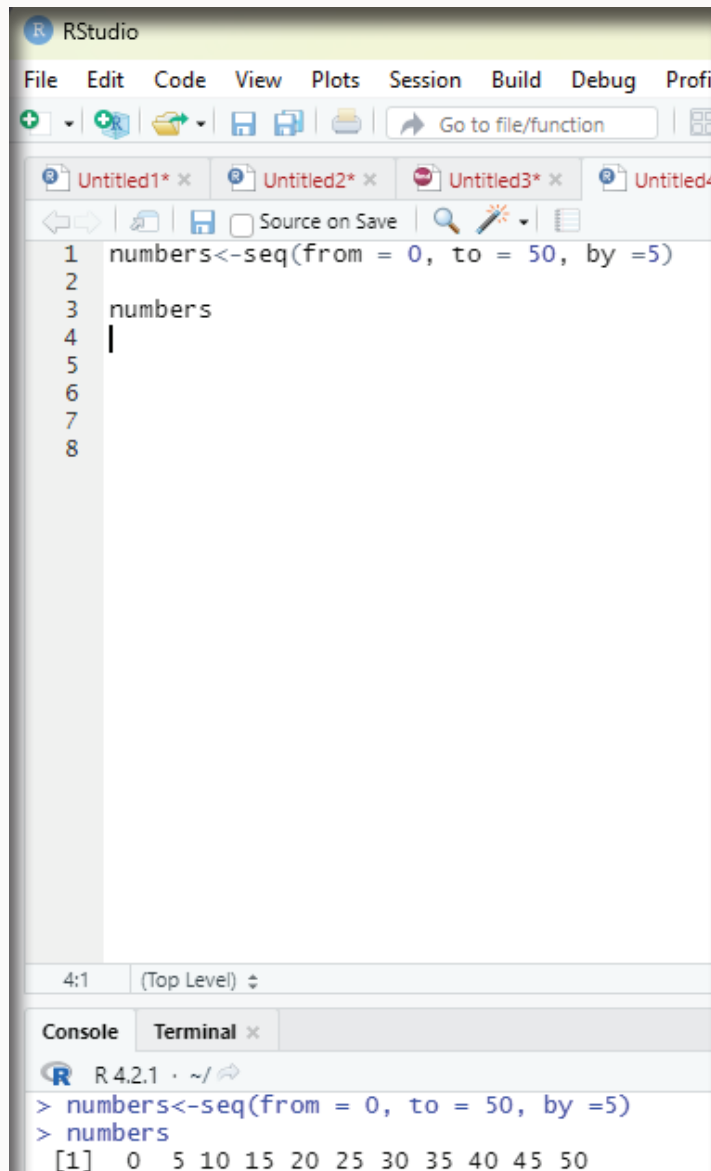


Image showing creation of sequenced vector

Example:

```
numbers<-seq(from = 0, to = 50, by =5)
```

```
numbers
```



The screenshot shows the RStudio interface. The source editor contains the following code:

```
1 numbers<-seq(from = 0, to = 50, by =5)
2
3 numbers
4 |
5
6
7
8
```

The console at the bottom shows the execution of the code:

```
> numbers<-seq(from = 0, to = 50, by =5)
> numbers
[1] 0 5 10 15 20 25 30 35 40 45 50
```

Image showing creation of a sequence of numbers with a difference of 5

List function:

A list in R can contain different data types inside it. A list in R is a collection of data that is ordered and can be changed.

In order to create a list, `list()` function is used.

Example:

```
# This is a list of strings.
```

```
thelist <-list("apple", "banana", "cherry")
```

```
# Print the list.
```

```
thelist
```

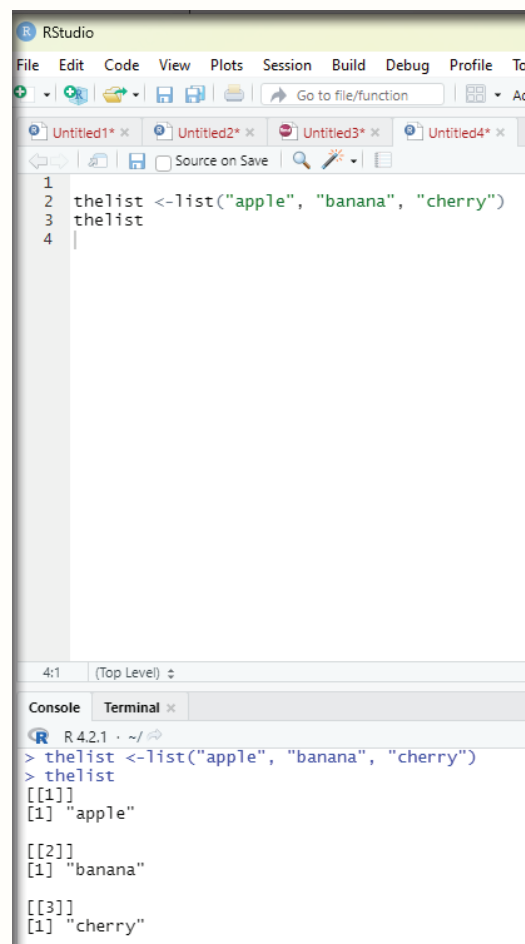


Image showing list function

Function to Access lists:

The user can access the list items by referring to its index number, inside brackets. The first item has an index number 1, the second has an index number of 2 and so on.

Example:

```
thelist <- list("apple", "banana", "cherry")  
thelist[1]
```

A screenshot of the RStudio interface. The top menu bar includes File, Edit, Code, View, Plots, Session, Build, Debug, Profile, and Tools. Below the menu is a toolbar with icons for file operations and a search bar. The editor window shows four tabs: Untitled1*, Untitled2*, Untitled3*, and Untitled4*. The active tab, Untitled3*, contains the following R code:

```
1 thelist <- list("apple", "banana", "cherry")  
2 thelist[1]  
3  
4
```

The status bar at the bottom indicates the cursor is at line 3, column 1, at the top level. Below the editor is a console window with the following output:

```
> thelist <- list("apple", "banana", "cherry")  
> thelist[1]  
[[1]]  
[1] "apple"
```

Image showing how to access list items using its index number

Changing item value:

In order to change the value of a specific item, it must be referred to by its index number.

Example:

Changing item value:

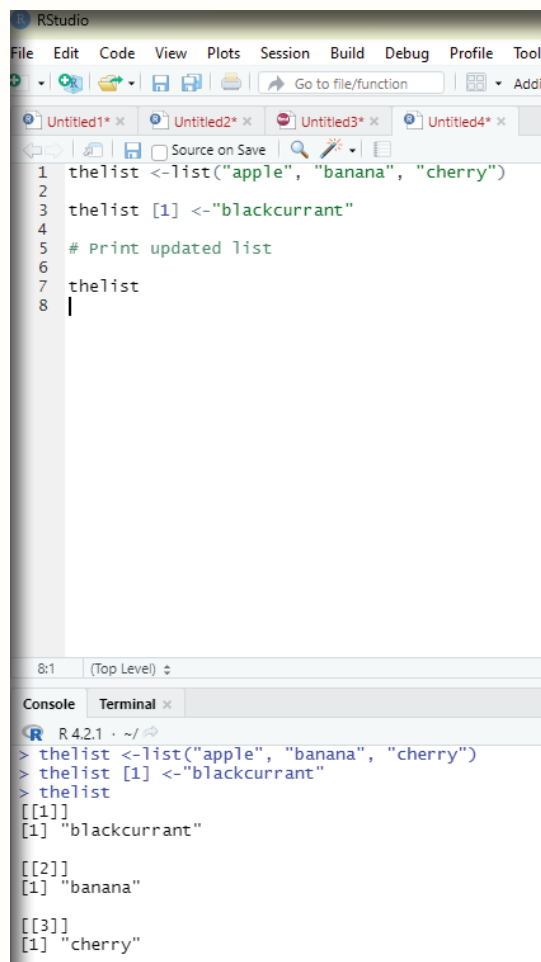
In order to change the value of a specific item, it must be referred to by its index number.

```
thelist <-list("apple", "banana", "cherry")
```

```
thelist [1] <- "blackcurrant"
```

Print updated list

```
thelist
```

A screenshot of the RStudio interface. The top pane shows a script with the following code:

```
1 thelist <-list("apple", "banana", "cherry")
2
3 thelist [1] <- "blackcurrant"
4
5 # Print updated list
6
7 thelist
8 |
```

The bottom pane shows the console output:

```
R 4.2.1 ~ /
> thelist <-list("apple", "banana", "cherry")
> thelist [1] <- "blackcurrant"
> thelist
[[1]]
[1] "blackcurrant"

[[2]]
[1] "banana"

[[3]]
[1] "cherry"
```

Image showing replacement of one item in the list with another

Function to ascertain the length of the list:

In order to find out how many items a list has, one has to use the `length()` function.

Example:

```
thelist <-list("apple", "banana", "cherry")  
length(thelist)
```

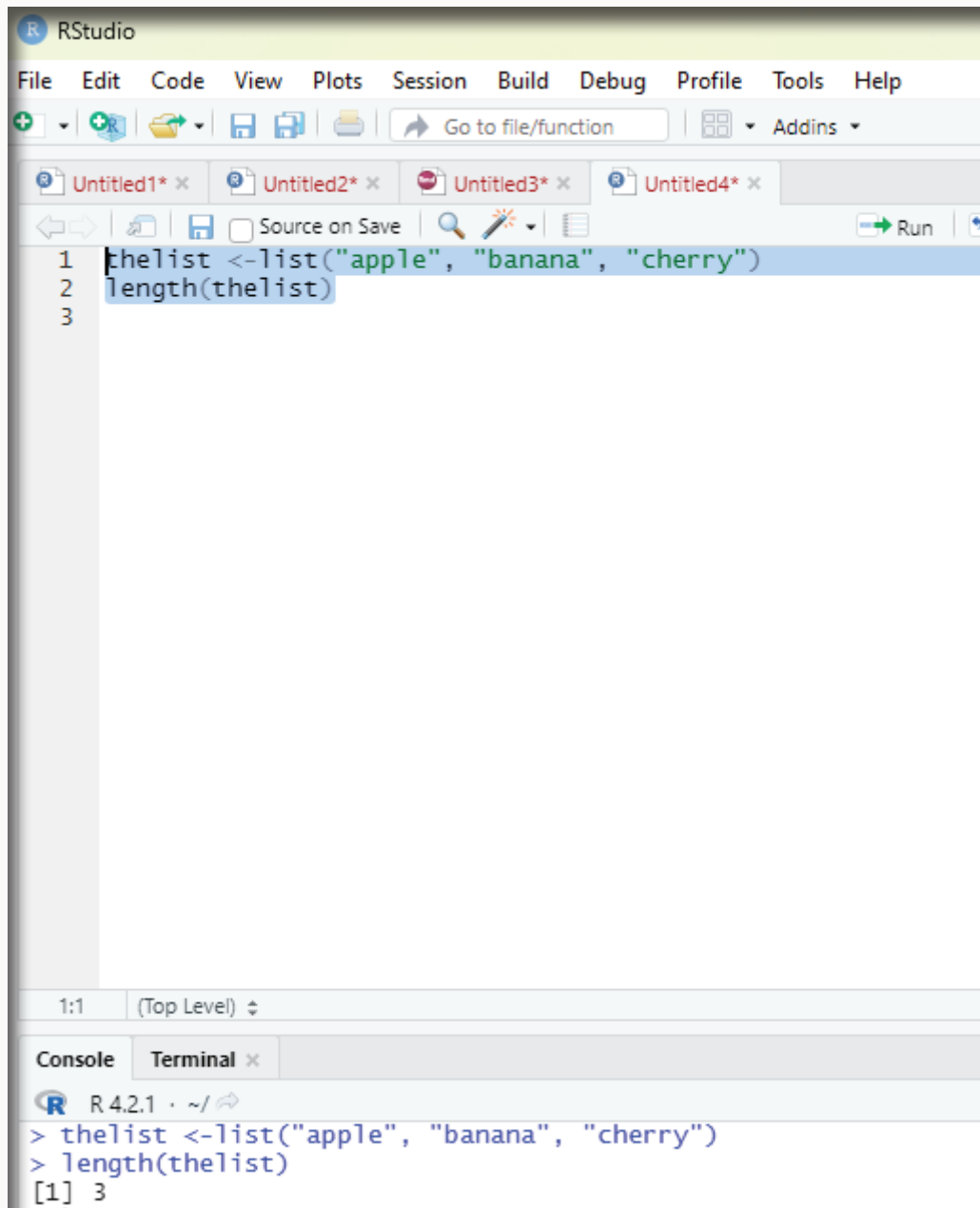


Image showing code for ascertaining the length of the list

In order to check if an item exists in the list the following function is to be used.

Operator to be used: %in%

Example:

```
thelist <-list("apple", "banana", "cherry")  
"apple" %in% thelist
```



The screenshot shows the RStudio interface. The script editor contains the following code:

```
1 thelist <-list("apple", "banana", "cherry")  
2 "apple" %in% thelist  
3  
4
```

The console at the bottom shows the execution of the code:

```
R 4.2.1 · ~/   
> thelist <-list("apple", "banana", "cherry")  
> "apple" %in% thelist  
[1] TRUE
```

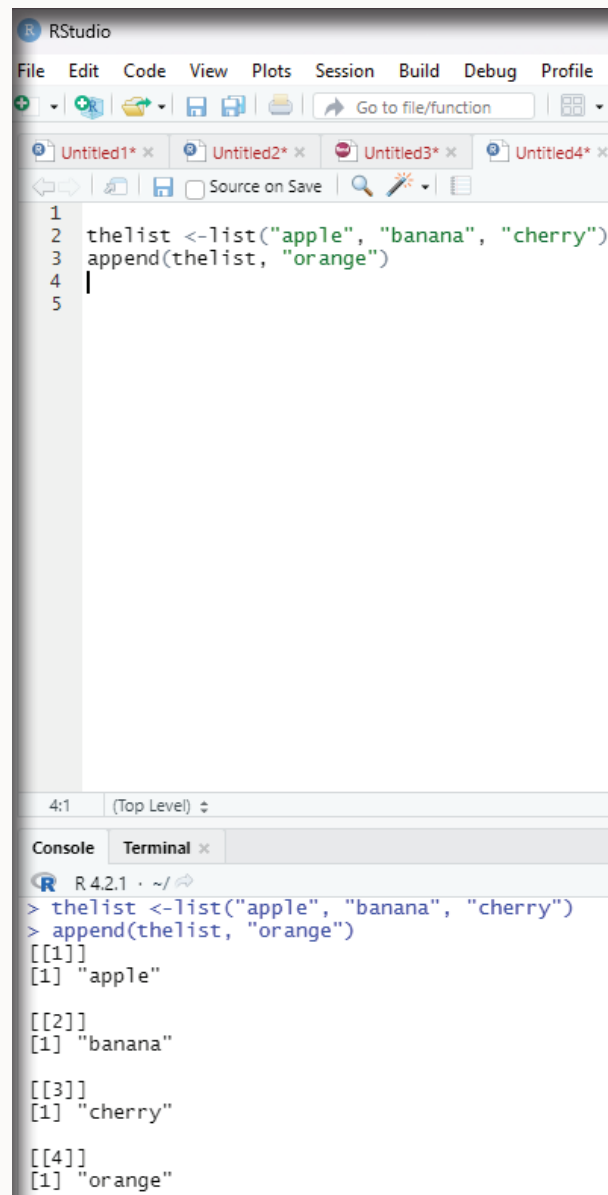
Image showing the code for ascertaining whether an item is present in the list or not in action

Adding list items:

To add an item to the end of the list, the user should use the `append()` function.

Example:

```
thelist <-list("apple", "banana", "cherry")  
append(thelist, "orange")
```



The screenshot shows the RStudio interface. The source editor contains the following R code:

```
1  
2 thelist <-list("apple", "banana", "cherry")  
3 append(thelist, "orange")  
4  
5
```

The console output shows the execution of the code:

```
R 4.2.1 ~/  
> thelist <-list("apple", "banana", "cherry")  
> append(thelist, "orange")  
[[1]]  
[1] "apple"  
  
[[2]]  
[1] "banana"  
  
[[3]]  
[1] "cherry"  
  
[[4]]  
[1] "orange"
```

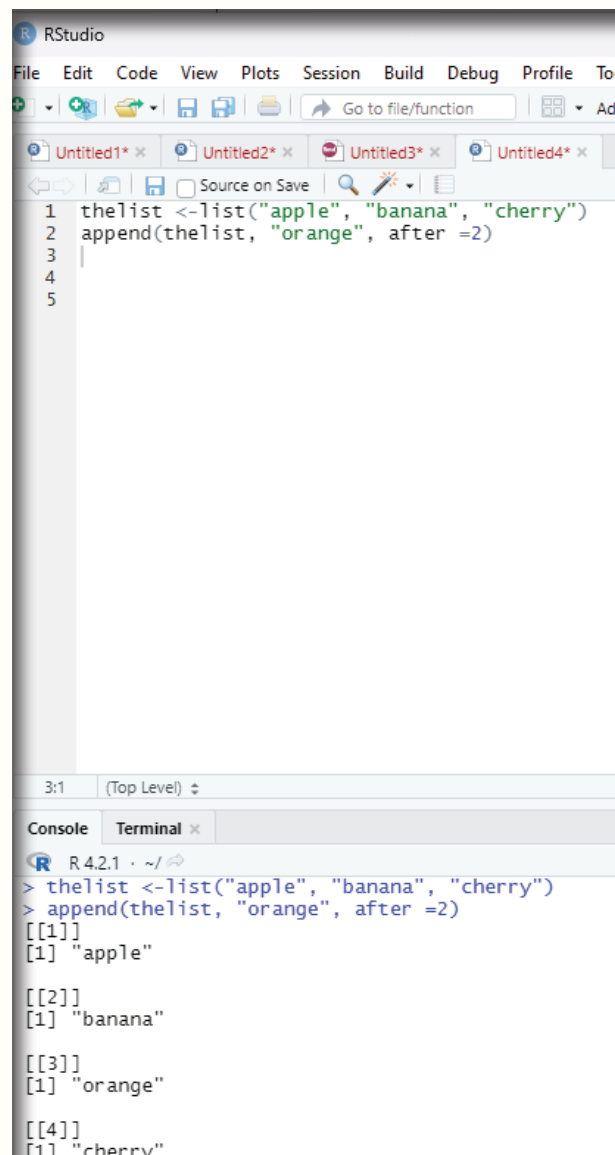
Image showing adding an item to the list

To add an item to the right of the specified index, add “after=index number” in the append() function.

Example:

To add “orange” to the list after “banana” (index2);

```
thelist <-list("apple", "banana", "cherry")  
append(thelist, "orange", after =2)
```



The screenshot shows the RStudio interface. The script editor contains the following code:

```
1 thelist <-list("apple", "banana", "cherry")  
2 append(thelist, "orange", after =2)  
3  
4  
5
```

The console output shows the execution of the code:

```
R 4.2.1 ~/  
> thelist <-list("apple", "banana", "cherry")  
> append(thelist, "orange", after =2)  
[[1]]  
[1] "apple"  
  
[[2]]  
[1] "banana"  
  
[[3]]  
[1] "orange"  
  
[[4]]  
[1] "cherry"
```

Image showing how to append an item after a specific item within a list

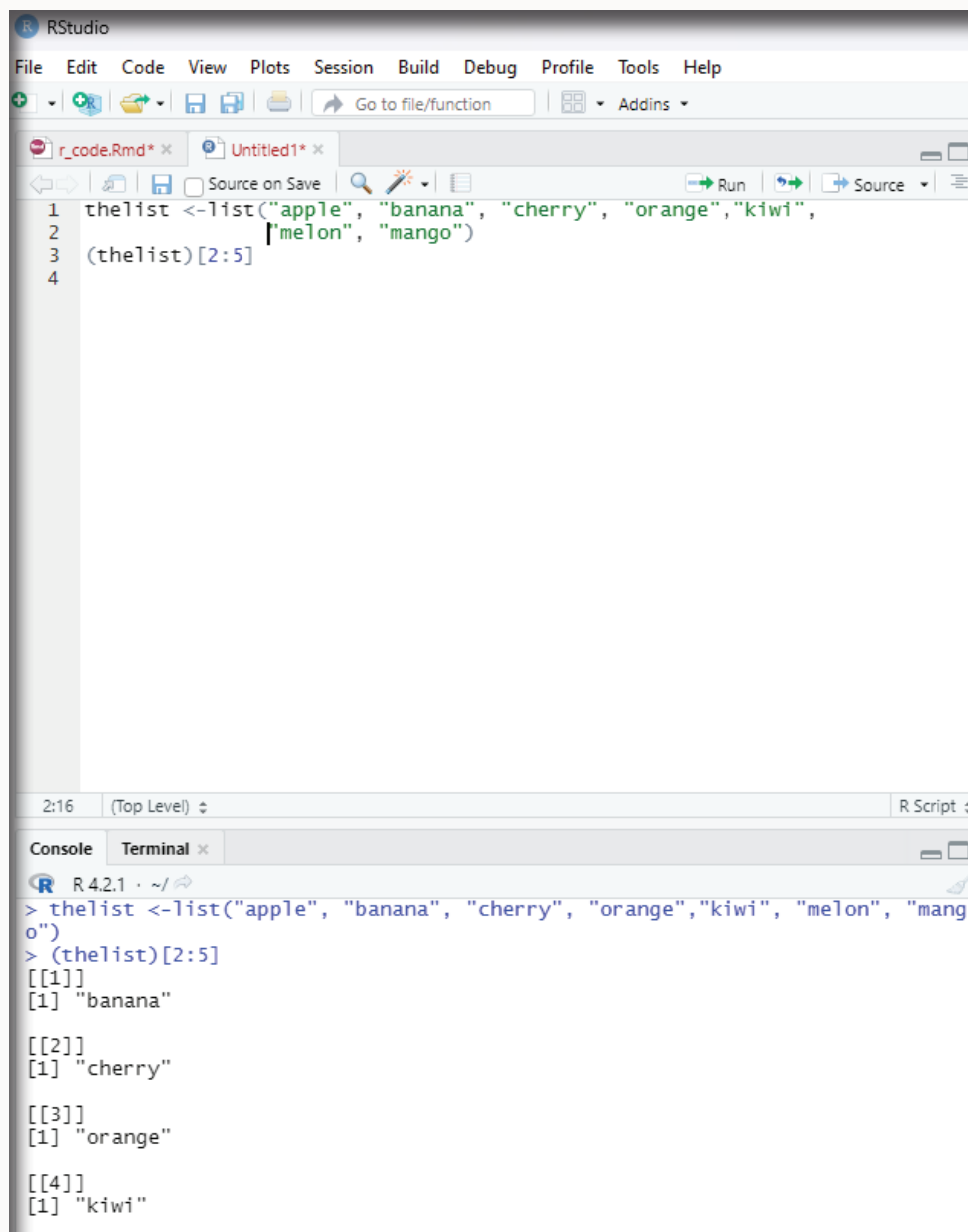
Removing list items:

The user can remove items from the list. The example code creates a new, updated list without “apple” by removing it from the list.

```
thelist <-list("apple", "banana", "cherry")  
thelist <-thelist[-1]
```

print the new list

thelist



The screenshot shows the RStudio interface. The script editor contains the following code:

```
1 thelist <-list("apple", "banana", "cherry", "orange", "kiwi",  
2               "melon", "mango")  
3 (thelist)[2:5]  
4
```

The console shows the execution of the code:

```
> thelist <-list("apple", "banana", "cherry", "orange", "kiwi", "melon", "mango")  
> (thelist)[2:5]  
[[1]]  
[1] "banana"  
  
[[2]]  
[1] "cherry"  
  
[[3]]  
[1] "orange"  
  
[[4]]  
[1] "kiwi"
```

Image showing range of indexes being specified

Range of indexes:

One can specify a range of indexes by specifying where to start and where to end the range by using : operator.

Example:

To return the second, third, fourth, and fifth item:

```
thelist <-list("apple", "banana", "cherry", "orange","kiwi", "melon", "mango")  
(thelist)[2:5]
```

Output:

```
[[1]]  
[1] "banana"
```

```
[[2]]  
[1] "cherry"
```

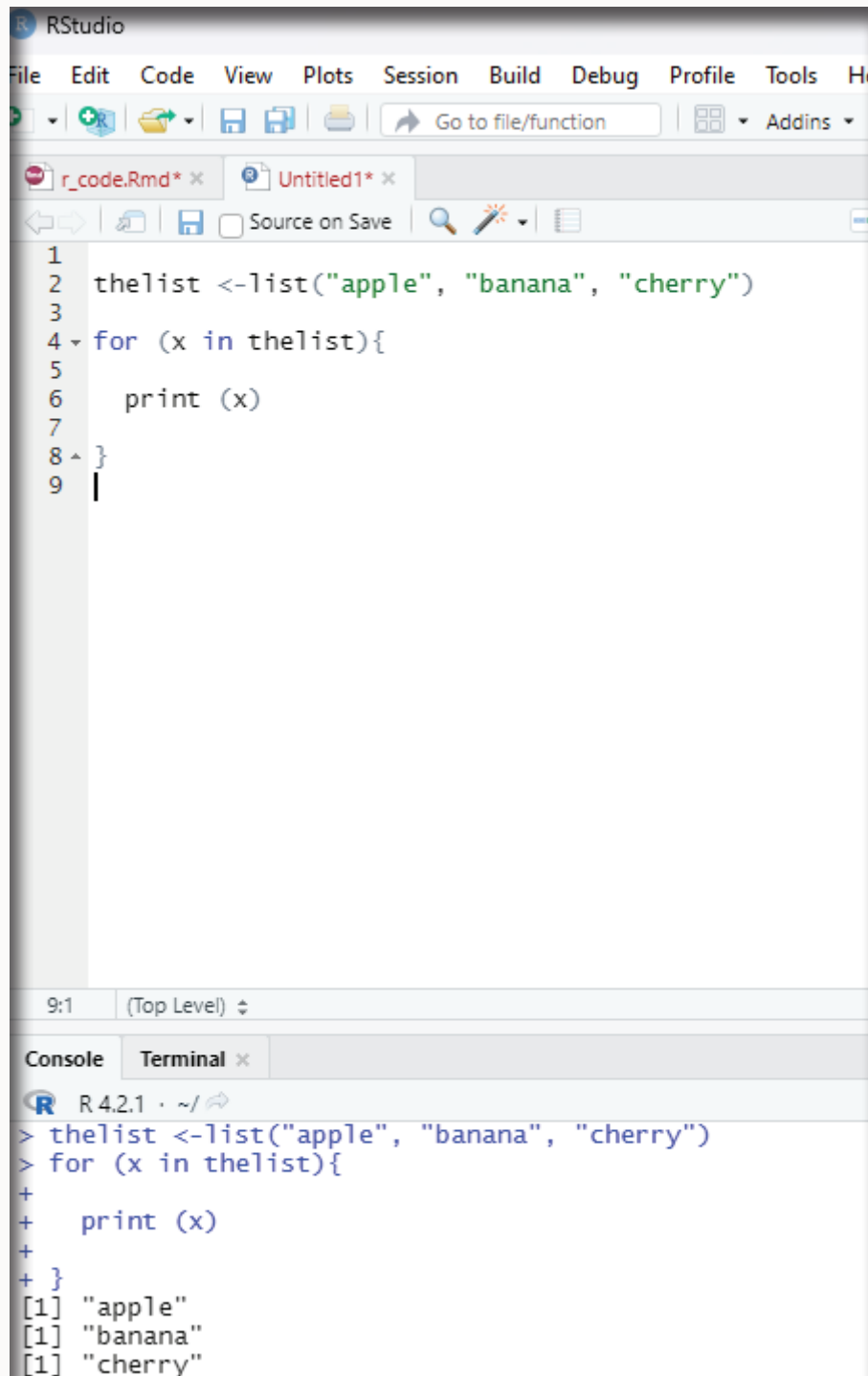
```
[[3]]  
[1] "orange"
```

```
[[4]]  
[1] "kiwi"
```

Loop through the list:

One can loop through the list items by using for loop:

```
thelist <-list("apple", "banana", "cherry")  
  
for (x in thelist){  
  
    print (x)  
  
}
```



The image shows the RStudio interface. The top menu bar includes File, Edit, Code, View, Plots, Session, Build, Debug, Profile, Tools, and Help. Below the menu is a toolbar with icons for file operations and a search bar labeled 'Go to file/function'. The editor pane shows a script with the following R code:

```
1  
2 thelist <-list("apple", "banana", "cherry")  
3  
4 for (x in thelist){  
5  
6   print (x)  
7  
8 }  
9 |
```

The status bar at the bottom of the editor shows '9:1' and '(Top Level)'.

The console pane at the bottom shows the execution of the code:

```
R 4.2.1 · ~/   
> thelist <-list("apple", "banana", "cherry")  
> for (x in thelist){  
+  
+   print (x)  
+  
+ }  
[1] "apple"  
[1] "banana"  
[1] "cherry"
```

Image showing loop function

In order to perform loop in R programming it is useful to iterate over the elements of a list, dataframe, vector, matrix or any other object. The loop can be used to execute a group of statements repeatedly depending upon the number of elements in the object. Loop is always entry controlled, where the test condition is tested first, then the body of the loop is executed. The loop body will not be executed if the test condition is false.

Syntax:

```
for(var in vector){  
  statements(s)  
}
```

In this syntax, var takes each value of the vector during the loop. In each iteration, the statements are evaluated.

Example for using for loop.

Iterating over a range in R - For loop.

```
# Use of for loop
```

```
for(i in 1:4)
```

```
{
```

```
  print (i^2)
```

```
}
```

Output:

```
[1] 1  
[1] 4  
[1] 9  
[1] 16
```

In the example above the ensures that the range of numbers between 1 to 4 inside a vector has been iterated and the resultant value displayed as the output.

Results demonstrate:

Multiplying each number by itself once:

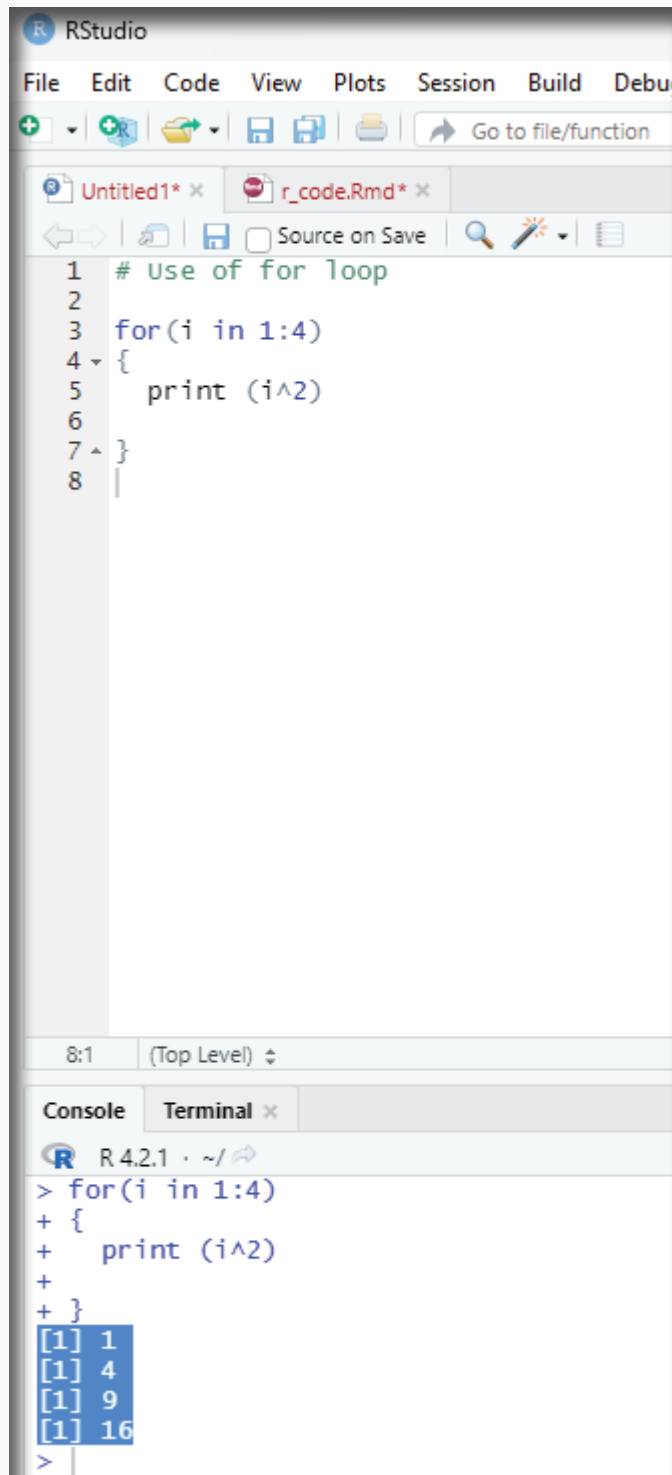
$1*1 = 1$

$2*2 = 2$

$3*3 = 9$

$4*4 = 16$

These resultant values are displayed as output.



The image shows the RStudio interface. The top menu bar includes File, Edit, Code, View, Plots, Session, Build, and Debug. Below the menu is a toolbar with icons for file operations and a search bar labeled 'Go to file/function'. The editor pane shows a script with the following code:

```
1 # Use of for loop
2
3 for(i in 1:4)
4 {
5   print (i^2)
6 }
7
8
```

The status bar at the bottom of the editor indicates '8:1 (Top Level)'. Below the editor is the Console pane, which shows the execution of the code:

```
> for(i in 1:4)
+ {
+   print (i^2)
+ }
[1] 1
[1] 4
[1] 9
[1] 16
>
```

Image showing loop function

Example using concatenate function in R - For loop.

Using concatenate outside the loop R - For loop:

R code to demonstrate the use of
for loop with vector

```
x <-c(-8,6,22,36)
for (i in x)
{
  print(i)
}
```

Output:

```
[1] -8
[1] 6
[1] 22
[1] 36
```

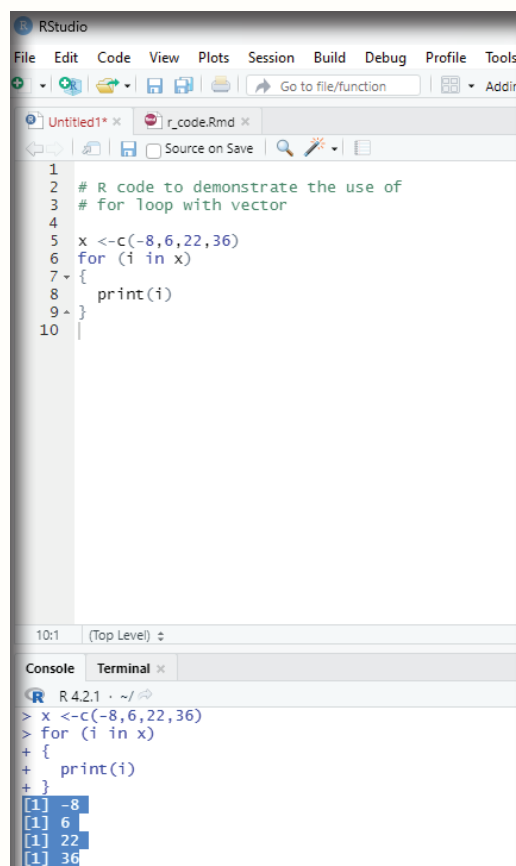


Image showing concatenate being used in loop function

Nested For-loop in R:

R language allows the use of one loop inside another one. For example, a for loop can be inside a while loop or vice versa.

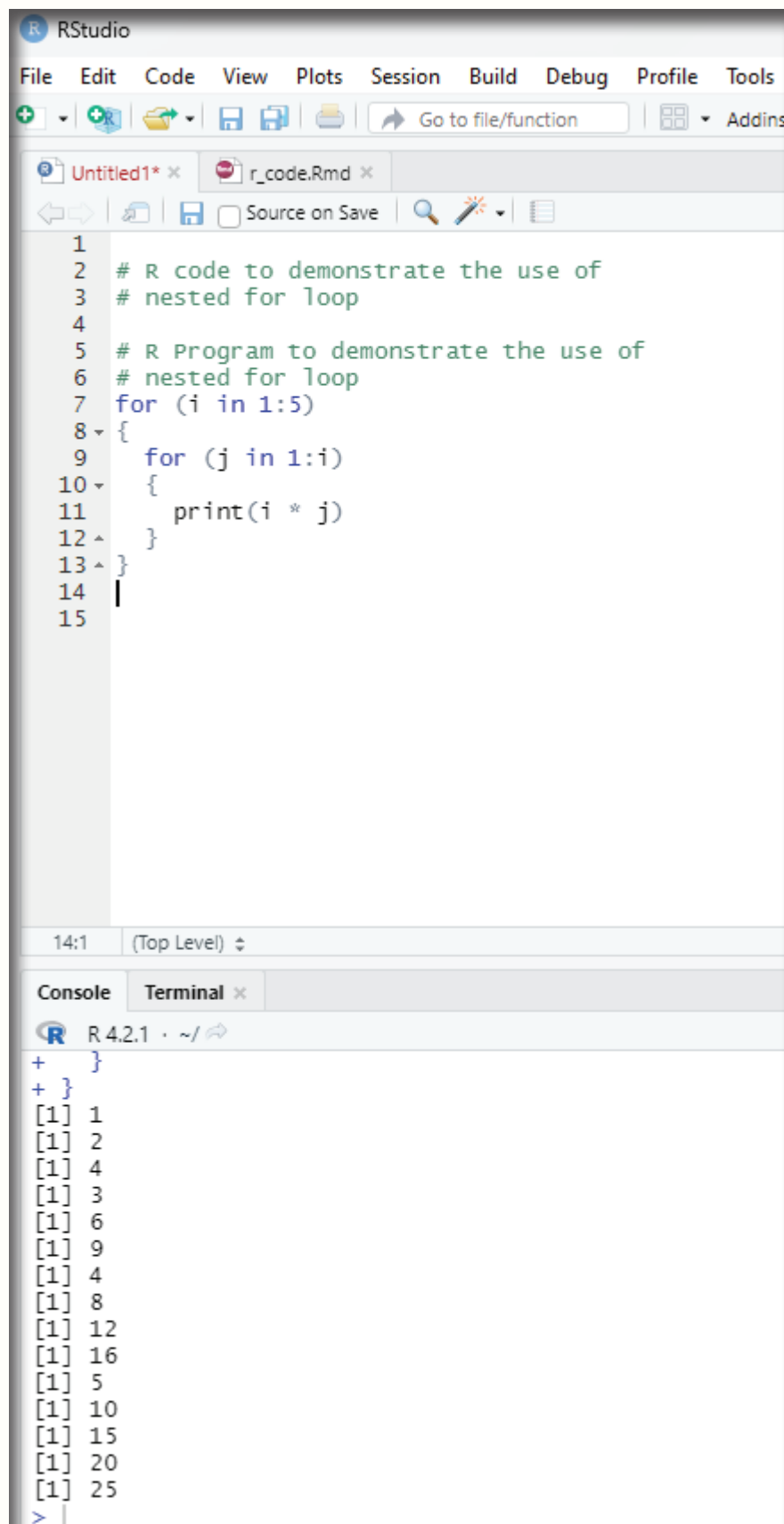
Example:

```
# R code to demonstrate the use of  
# nested for loop
```

```
# R Program to demonstrate the use of  
# nested for loop  
for (i in 1:5)  
{  
  for (j in 1:i)  
  {  
    print(i * j)  
  }  
}
```

Output:

```
[1] 1  
[1] 2  
[1] 4  
[1] 3  
[1] 6  
[1] 9  
[1] 4  
[1] 8  
[1] 12  
[1] 16  
[1] 5  
[1] 10  
[1] 15  
[1] 20  
[1] 25
```



The image shows the RStudio interface. The top menu bar includes File, Edit, Code, View, Plots, Session, Build, Debug, Profile, and Tools. Below the menu is a toolbar with icons for file operations and a search bar labeled 'Go to file/function'. The editor pane shows a script with the following R code:

```
1  
2 # R code to demonstrate the use of  
3 # nested for loop  
4  
5 # R Program to demonstrate the use of  
6 # nested for loop  
7 for (i in 1:5)  
8 {  
9   for (j in 1:i)  
10  {  
11    print(i * j)  
12  }  
13 }  
14 |  
15
```

The console pane at the bottom shows the output of the code, which is a sequence of numbers from 1 to 25, each preceded by '[1]'. The output is as follows:

```
R 4.2.1 ~/  
+ }  
+ }  
[1] 1  
[1] 2  
[1] 4  
[1] 3  
[1] 6  
[1] 9  
[1] 4  
[1] 8  
[1] 12  
[1] 16  
[1] 5  
[1] 10  
[1] 15  
[1] 20  
[1] 25  
>
```

Image showing nested loop

Jump statements in R:

One can use jump statements in loops to terminate the loop at a particular iteration or to skip a particular iteration in the loop. The two most commonly used jump statements are:

Break statement:

This type of jump statement is used to terminate the loop at a particular iteration. The program then continues with the next statement outside the loop if any.

Example:

```
# This code is to demonstrate the use of break in for loop.
```

```
for (i in c(4,5,34,22,12,9))
```

```
{
```

```
  if (i == 0)
```

```
  {
```

```
    break
```

```
  }
```

```
  print(i)
```

```
}
```

```
print("Outside Loop")
```

Output:

```
[1] 4
```

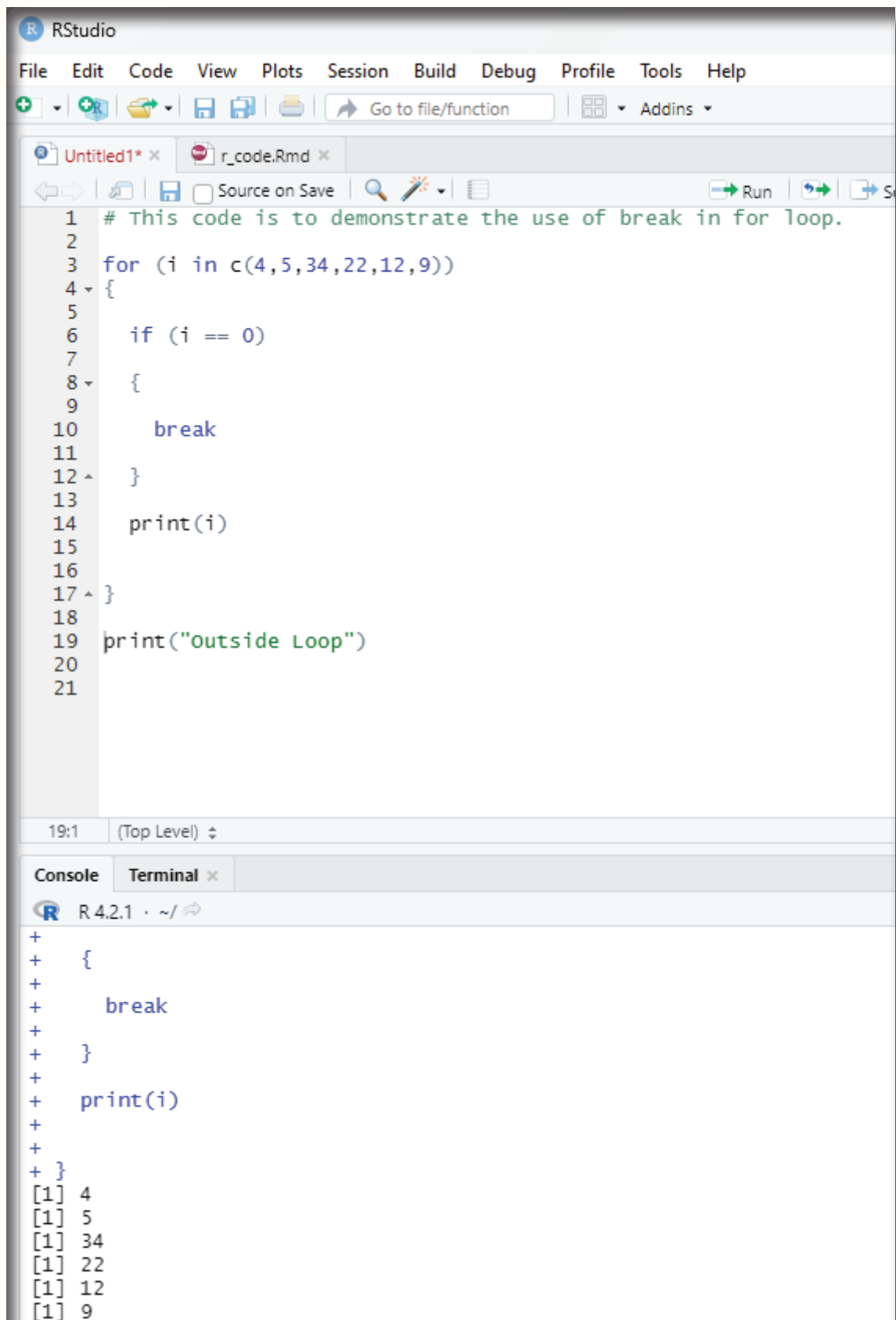
```
[1] 5
```

```
[1] 34
```

```
[1] 22
```

```
[1] 12
```

```
[1] 9
```



```
1 # This code is to demonstrate the use of break in for loop.
2
3 for (i in c(4,5,34,22,12,9))
4 {
5
6   if (i == 0)
7   {
8     break
9   }
10  print(i)
11
12 }
13
14 print("Outside Loop")
15
16
17
18
19
20
21
```

19:1 (Top Level) ↕

Console Terminal x

R 4.2.1 · ~/

```
+ {
+   break
+ }
+ print(i)
+
+ }
[1] 4
[1] 5
[1] 34
[1] 22
[1] 12
[1] 9
```

Image showing loop with break statement

Matrices:

This is a two dimensional data set with columns and rows.

A column is a vertical representation of data, while a row is a horizontal representation of data.

A matrix can be treated with `matrix()` function. Specify the `nrow` and `ncol` parameters to get the amount of rows and columns:

```
# Creating a matrix
```

```
thematrix <- matrix(c(1,2,3,4,5,6), nrow =3, ncol = 2)
```

```
# Print the matrix
```

```
thematrix
```

Output:

```
[,1] [,2]  
[1,] 1  4  
[2,] 2  5  
[3,] 3  6
```

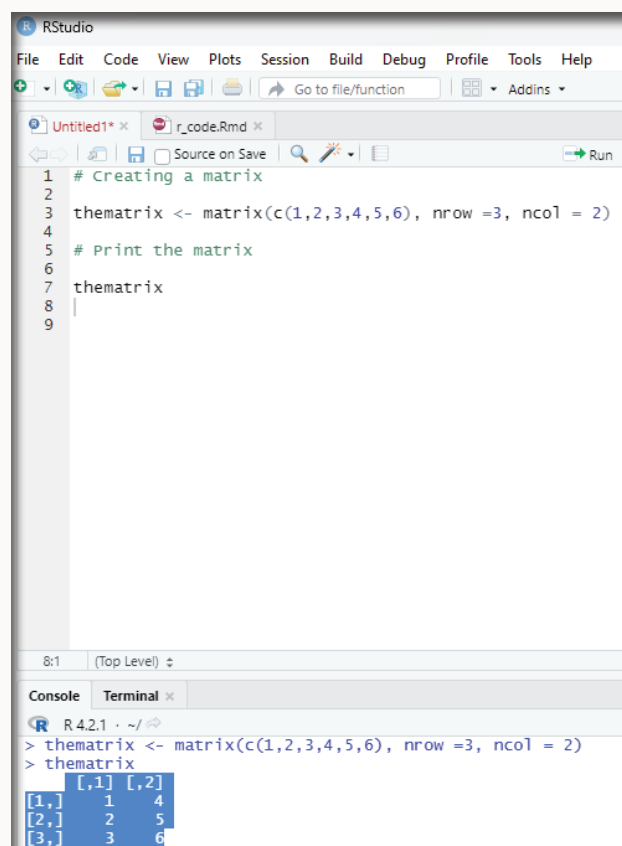


Image showing matrix creation

Accessing matrix items:

The user can access the items by using []. The first number "1" in the bracket specifies the row position, while the second number "2" specifies column position.

Example:

```
fruitmatrix <- matrix(c("apple", "banana", "cherry", "orange"), nrow = 2, ncol = 2)
```

```
fruitmatrix[1,2]
```

Output:

[1] "cherry"

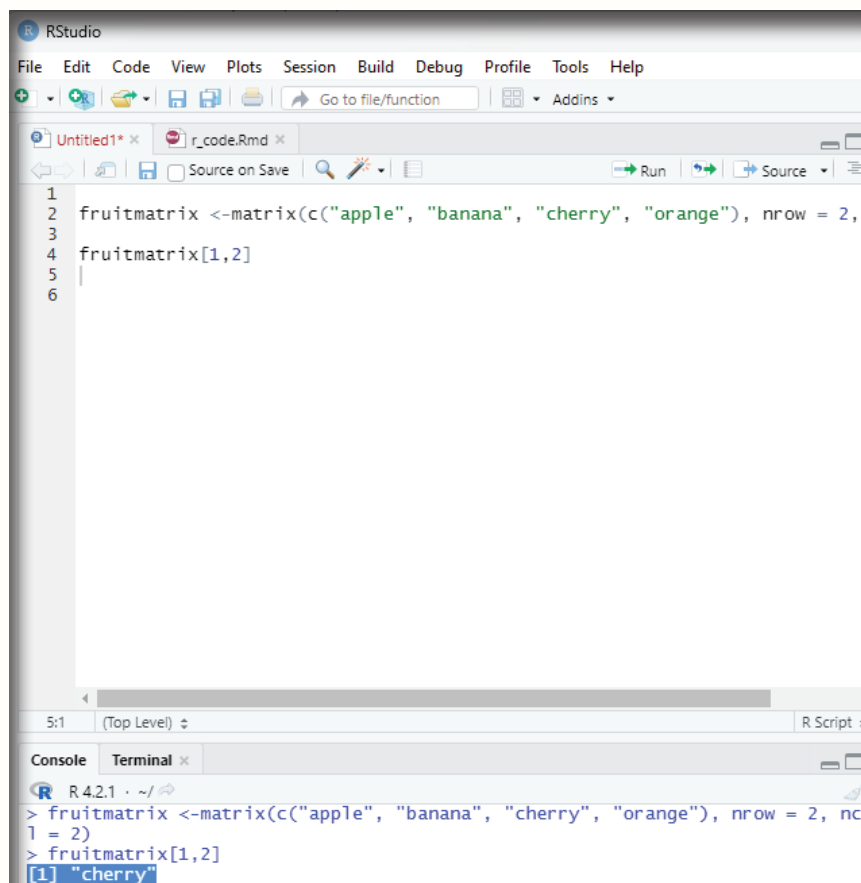


Image showing matrix items being accessed

The whole row can be accessed if the user specifies a comma after the number in the bracket.

```
fruitmatrix[ 2,]
```

the whole column can be accessed if the user specifies a comma before the number in the bracket.

```
fruitmatrix[,2]
```

In a matrix more than one row can be accessed using c() function.

Example:

```
fruitmatrix <-matrix (c ("apple", "orange", "Papaya", "pineapple", "pear", "grapes", "seetha", "ba-  
nana", "sapota"), nrow =3, ncol=3)
```

```
fruitmatrix[c(1,2),]
```

More than one column can be accessed by using c() function.

```
fruitmatrix [,c(1,2)]
```

Adding rows and columns to a matrix:

For this purpose cbind() function can be used.

Example:

```
fruitmatrix <- matrix(c("apple", "banana", "cherry", "orange", "grape", "pineapple", "pear", "mel-  
on", "fig"),  
nrow =3, ncol=3)
```

```
newfruitmatrix <-cbind(fruitmatrix,c("strawberry", "blueberry", "raspberry"))
```

```
#print the new matrix
```

```
newfruitmatrix
```

It should be noted that the new column should be of the same length as the existing matrix.

```
thismatrix <- matrix(c("apple", "banana", "cherry", "orange", "grape", "pineapple", "pear", "melon", "fig"), nrow = 3, ncol = 3)
```

```
newmatrix <- rbind(thismatrix, c("strawberry", "blueberry", "raspberry"))
```

```
# Print the new matrix
```

```
newmatrix
```

Removing rows and columns:

To remove rows and columns c() function can be used.

Example:

```
fruitmatrix <- matrix(c("apple", "banana", "pear", "orange", "sapota", "papaya"), nrow=3, ncol=2)
```

```
#Remove the first row and the first column.
```

```
fruitmatrix <- fruitmatrix[-c(1), -c(1)]
```

```
fruitmatrix
```

Checking if an item is present in the matrix. For this purpose %in% operator can be used.

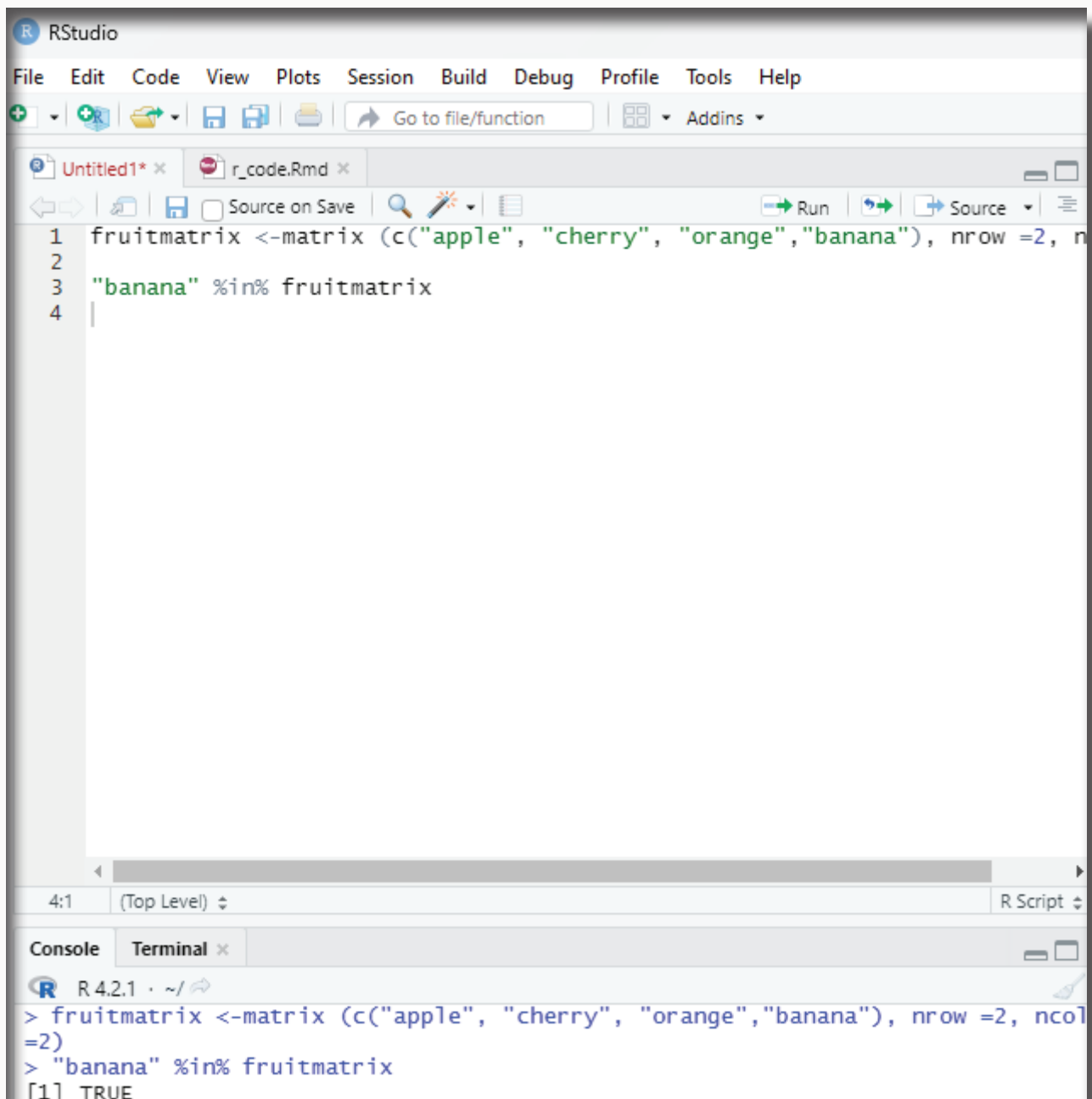
Example:

This code is to check if "banana" is present in the matrix.

```
fruitmatrix <- matrix(c("apple", "cherry", "orange", "banana"), nrow = 2, ncol = 2)
```

```
"banana" %in% fruitmatrix
```

If it is present the output generated will state True.



The image shows the RStudio interface. The script editor contains the following R code:

```
1 fruitmatrix <-matrix (c("apple", "cherry", "orange","banana"), nrow =2, ncol =2)
2
3 "banana" %in% fruitmatrix
4
```

The console shows the execution of the code:

```
> fruitmatrix <-matrix (c("apple", "cherry", "orange","banana"), nrow =2, ncol =2)
> "banana" %in% fruitmatrix
[1] TRUE
```

Image showing query for identifying whether banana is present in the matrix

Number of rows and columns can be found by using dim() function.

```
fruitmatrix <- matrix(c("apple", "pear", "banana", "dragonfruit"), nrow = 2, ncol = 2)
```

```
dim(fruitmatrix)
```

Length of the matrix.

length() function can be used to find the dimension of a matrix.

Example:

```
fruitmatrix <- matrix(c("apple", "pear", "banana", "dragonfruit"), nrow = 2, ncol = 2)
```

```
length(fruitmatrix)
```

This value is actually the total number of cells in a matrix. (number of rows multiplied by the number of columns).

Loop through the matrix:

The user can loop through a matrix by using for loop. The loop starts at the first row, moves to the right.

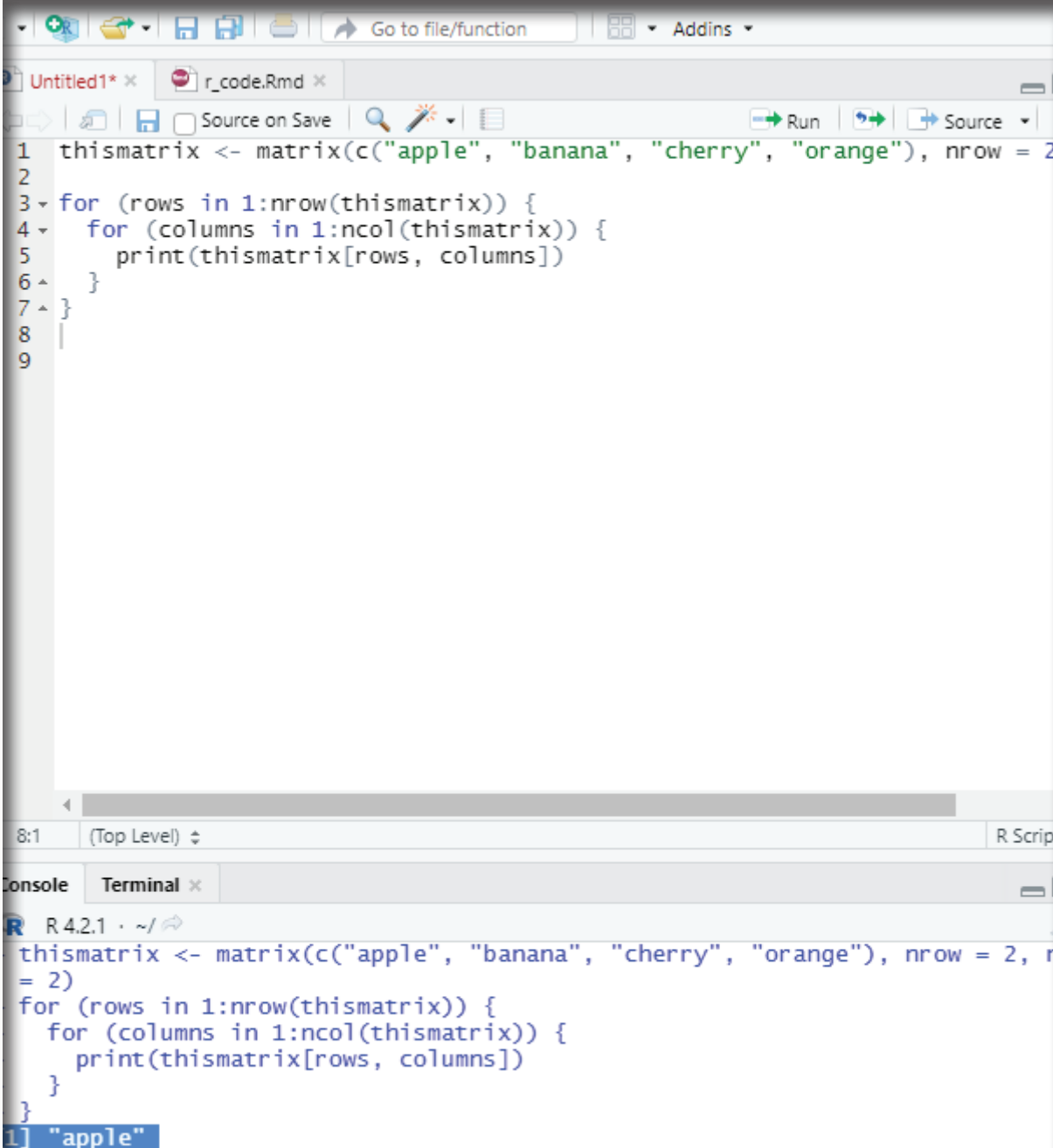
Example:

```
thismatrix <- matrix(c("apple", "banana", "cherry", "orange"), nrow = 2, ncol = 2)
```

```
for (rows in 1:nrow(thismatrix)) {  
  for (columns in 1:ncol(thismatrix)) {  
    print(thismatrix[rows, columns])  
  }  
}
```

Output:

```
[1] "apple"  
[1] "cherry"  
[1] "banana"  
[1] "orange"
```



The screenshot shows an RStudio interface with a script editor and a console. The script editor contains the following R code:

```
1 thismatrix <- matrix(c("apple", "banana", "cherry", "orange"), nrow = 2, ncol = 4)  
2  
3 for (rows in 1:nrow(thismatrix)) {  
4   for (columns in 1:ncol(thismatrix)) {  
5     print(thismatrix[rows, columns])  
6   }  
7 }  
8  
9
```

The console shows the output of the script, which is the first element of the matrix, "apple", highlighted in blue.

```
R 4.2.1 ~/  
thismatrix <- matrix(c("apple", "banana", "cherry", "orange"), nrow = 2, ncol = 4)  
for (rows in 1:nrow(thismatrix)) {  
  for (columns in 1:ncol(thismatrix)) {  
    print(thismatrix[rows, columns])  
  }  
}  
[1] "apple"
```

Image showing loop function in matrix

Example:

```
# Combine matrices
```

```
Matrix1 <- matrix(c("apple", "banana", "cherry", "grape"), nrow = 2, ncol = 2)
```

```
Matrix2 <- matrix(c("orange", "mango", "pineapple", "watermelon"), nrow = 2, ncol = 2)
```

```
# Adding it as a rows
```

```
Matrix_Combined <- rbind(Matrix1, Matrix2)
```

```
Matrix_Combined
```

```
# Adding it as a columns
```

```
Matrix_Combined <- cbind(Matrix1, Matrix2)
```

```
Matrix_Combined
```

Arrays:

Arrays can have more than two dimensions, this is the difference between matrix (one dimensional array) and array.

One can use the array() function to create an array, and the dim parameter to specify the dimensions.

Example:

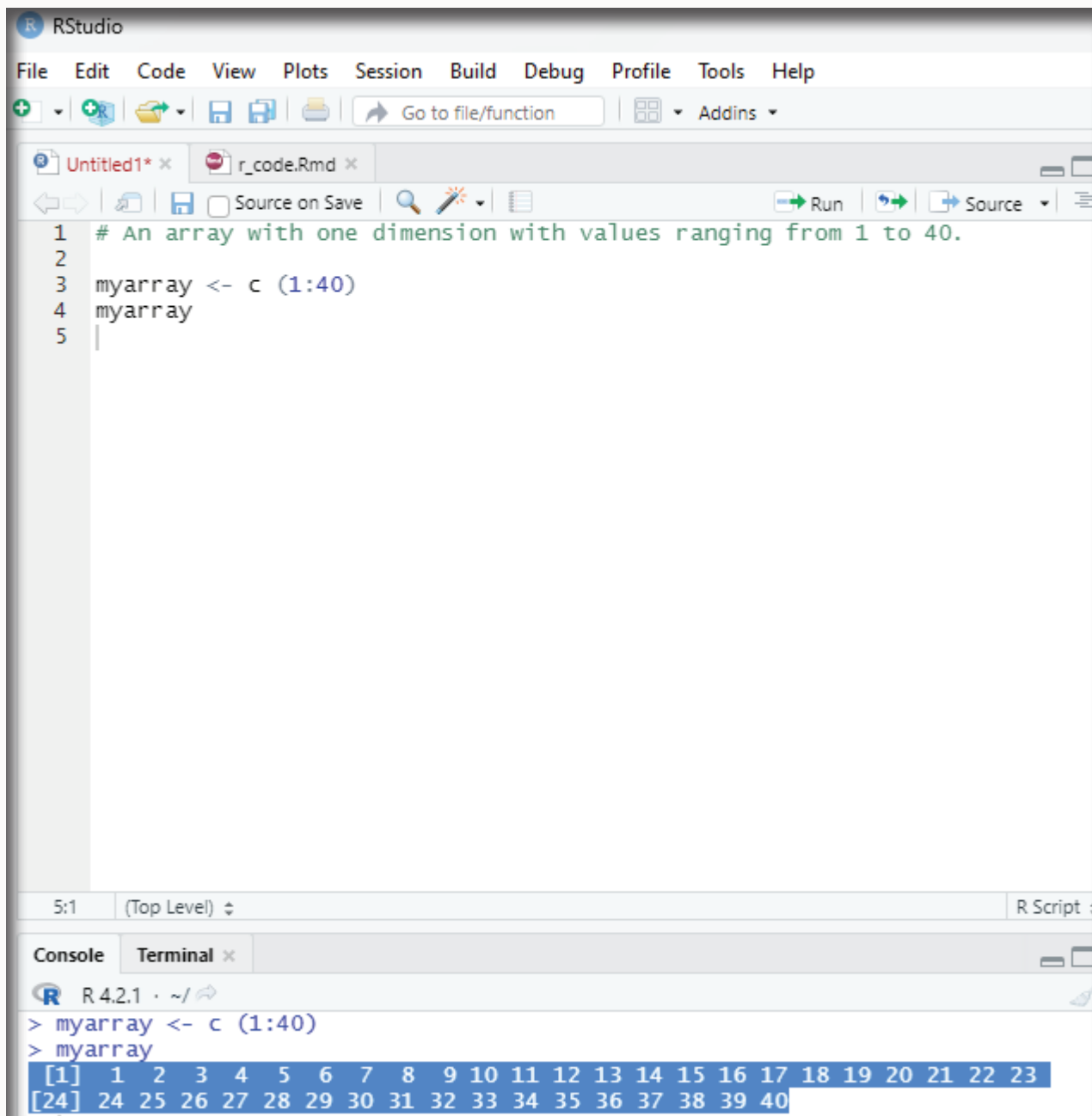
```
# An array with one dimension with values ranging from 1 to 40.
```

```
myarray <- c(1:40)
```

```
myarray
```

Output:

```
[1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23  
[24] 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40
```



The image shows the RStudio interface. The top menu bar includes File, Edit, Code, View, Plots, Session, Build, Debug, Profile, Tools, and Help. Below the menu is a toolbar with icons for file operations and a search bar. The main editor window displays a script with the following code:

```
1 # An array with one dimension with values ranging from 1 to 40.  
2  
3 myarray <- c (1:40)  
4 myarray  
5
```

The bottom panel shows the Console window with the following output:

```
> myarray <- c (1:40)  
> myarray  
[1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23  
[24] 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40
```

Image showing array formation

An array with more than one dimension.

```
multiarray <-array(myarray,dim = c(4,3,2))
```

```
multiarray
```

In the above example code it creates an array with values from 1 to 40.

Explanation for dim=c(4,3,2). The first and second number specifies the the number of rows and columns and the last number within the bracket specifies the number of dimensions needed.

Note: Arrays can have only one data type.

Output:

```
[,1] [,2] [,3]
[1,]  1  5  9
[2,]  2  6 10
[3,]  3  7 11
[4,]  4  8 12
```

```
,, 2
```

```
[,1] [,2] [,3]
[1,] 13 17 21
[2,] 14 18 22
[3,] 15 19 23
[4,] 16 20 24
```

Access Array items:

The user can access the elements within an array by referring to their index position. One can use the [] brackets to access the desired elements in the array.

Access all the items from the first row from matrix one.

```
multiarray <-array(myarray, dim = c(4,3,2))
```

```
multiarray [c(1),,1]
```

Access all the items from the first column from matrix one.

```
multiarray <-array(myarray, dim=c(4,3,2))
```

```
multiarray [,c(1),1]
```

Output:

```
[1] 1 5 9
```

Explanation:

A comma (,) before c() means that the user wants to access the column.

A comma (,) after c() means that the user wants to access the row.

Code to check if an item exists in an array:

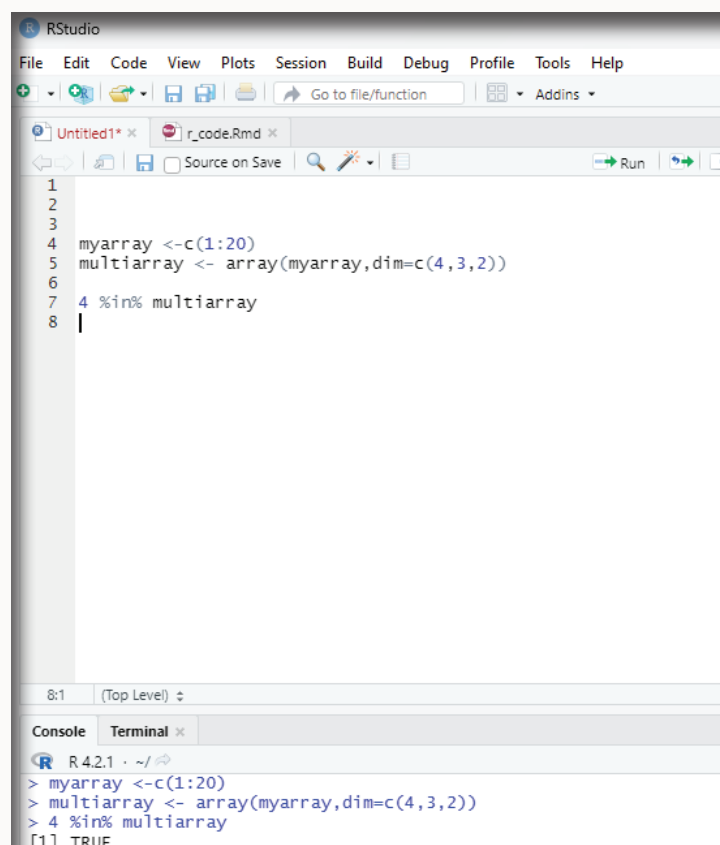
In order to find out if a specified item is present in an array one can use %in% operator.

To check if the value of 4 is present in the array.

```
myarray <-c(1:20)
```

```
multiarray <- array(myarray,dim=c(4,3,2))
```

```
4 %in% multiarray
```

A screenshot of the RStudio interface. The top menu bar includes File, Edit, Code, View, Plots, Session, Build, Debug, Profile, Tools, and Help. Below the menu is a toolbar with icons for file operations and a 'Go to file/function' search bar. The main editor window shows a script with the following code:

```
1  
2  
3  
4 myarray <-c(1:20)  
5 multiarray <- array(myarray,dim=c(4,3,2))  
6  
7 4 %in% multiarray  
8 |
```

The bottom panel shows the 'Console' tab with the following output:

```
R 4.2.1 ~/  
> myarray <-c(1:20)  
> multiarray <- array(myarray,dim=c(4,3,2))  
> 4 %in% multiarray  
[1] TRUE
```

Image showing the code to verify if 4 is present in the array

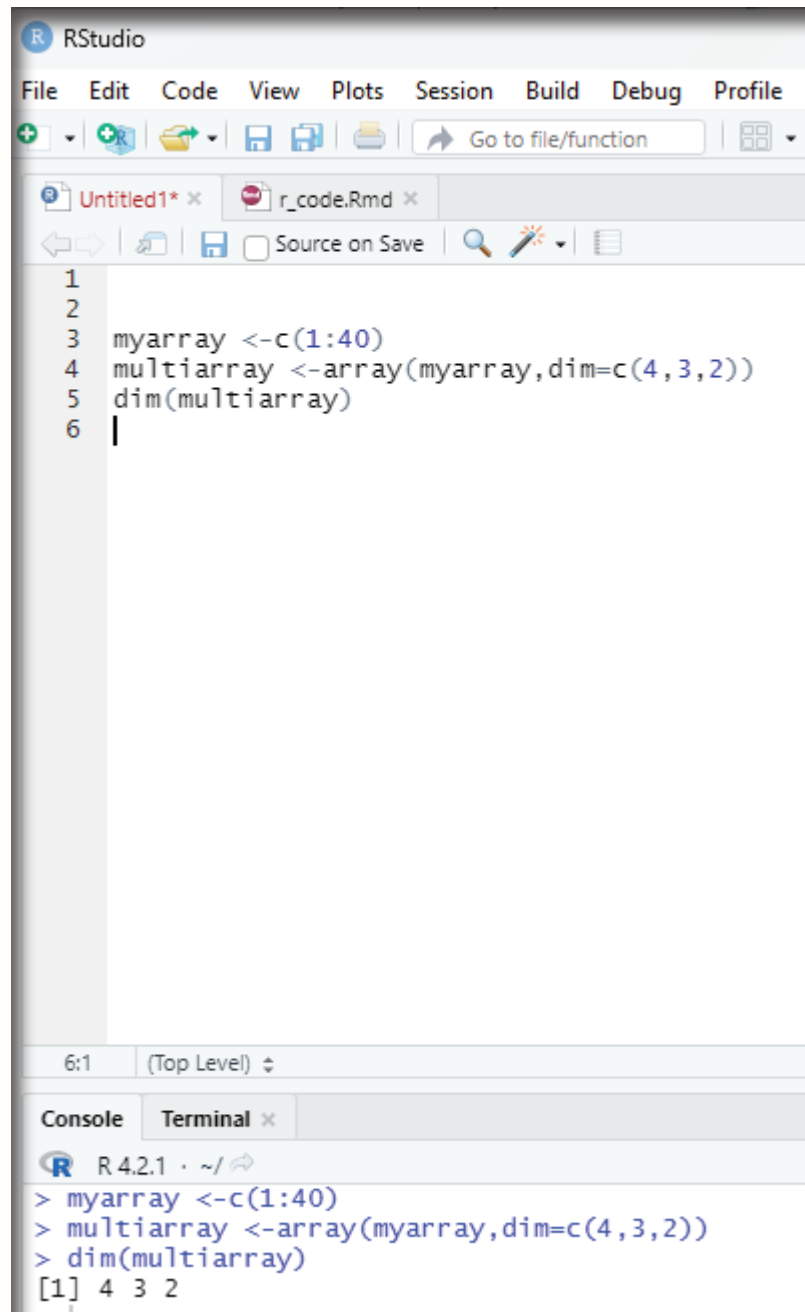
To calculate the number of Rows and columns `dim()` function can be used.

Example:

```
myarray <-c(1:40)
```

```
multiarray <-array(myarray,dim=c(4,3,2))
```

```
dim(multiarray)
```



The screenshot shows the RStudio interface. The script editor contains the following code:

```
1  
2  
3 myarray <-c(1:40)  
4 multiarray <-array(myarray,dim=c(4,3,2))  
5 dim(multiarray)  
6 |
```

The console at the bottom shows the execution of the code:

```
R 4.2.1 · ~/ |  
> myarray <-c(1:40)  
> multiarray <-array(myarray,dim=c(4,3,2))  
> dim(multiarray)  
[1] 4 3 2
```

Image showing code for calculating the number of rows and columns

Array length.

To calculate the length of the array `length()` function can be used.

`length(multiarray)`

Loop through an array:

One can loop through the array items by using a for loop: function.

Example:

```
myarray <-c(1:20)
multiarray <-array(myarray, dim = c(4,3,2))

for (x in multiarray){
  print(x)
}
```

R Factors:

Factors are used to categorize data. Examples of factors include:

Demography: Male/Female

Music: Rock, Pop, Classic

Training: Strength, Stamina

Example:

Create a factor.

```
music_genre <-factor (c("jazz", "Rock", "Classic","Hindustani", "Carnatic"))

#Print the factor

music_genre
```

Data Entry in R Programming

R or RStudio by default does not open up a spread sheet interface on execution. This is because in R one needs to approach data a little differently by writing out each step in code. R does have a spreadsheet like data entry tool.

Everything in R is an object and this is the basic difference between R and Excel. For this to happen the user needs to set up a blank data frame (similar to that of Excel table with rows and columns). If the user leaves the arguments blank in `data.frame` it would result in creation of an empty data frame.

Code:

```
myData <- data.frame()
```

This code on execution will create an empty data frame. This command will still not launch the viewer. For entering the data into the data frame the command to edit data in the viewer should be invoked.

Code:

```
myData <- edit(myData)
```

On running this command the Data Editor launches.

The default names of the column can be changed by clicking on top of it. While entering the data, the data editor gives the user the option of specifying the type of data entered. On closing the editor the data gets saved and the editor closes. The entered data can be checked by invoking the print data command.

One flip side of data editor is that it does not set the column to logical when logical values are entered. The entire column should be converted to logical using the following command in the scripting window.

```
myData # This command will open the data editor window.
```

```
is.logical(nameofdatafile&Isnameofthecolumn)
```

Data can be entered from within the scripting window using command functions.

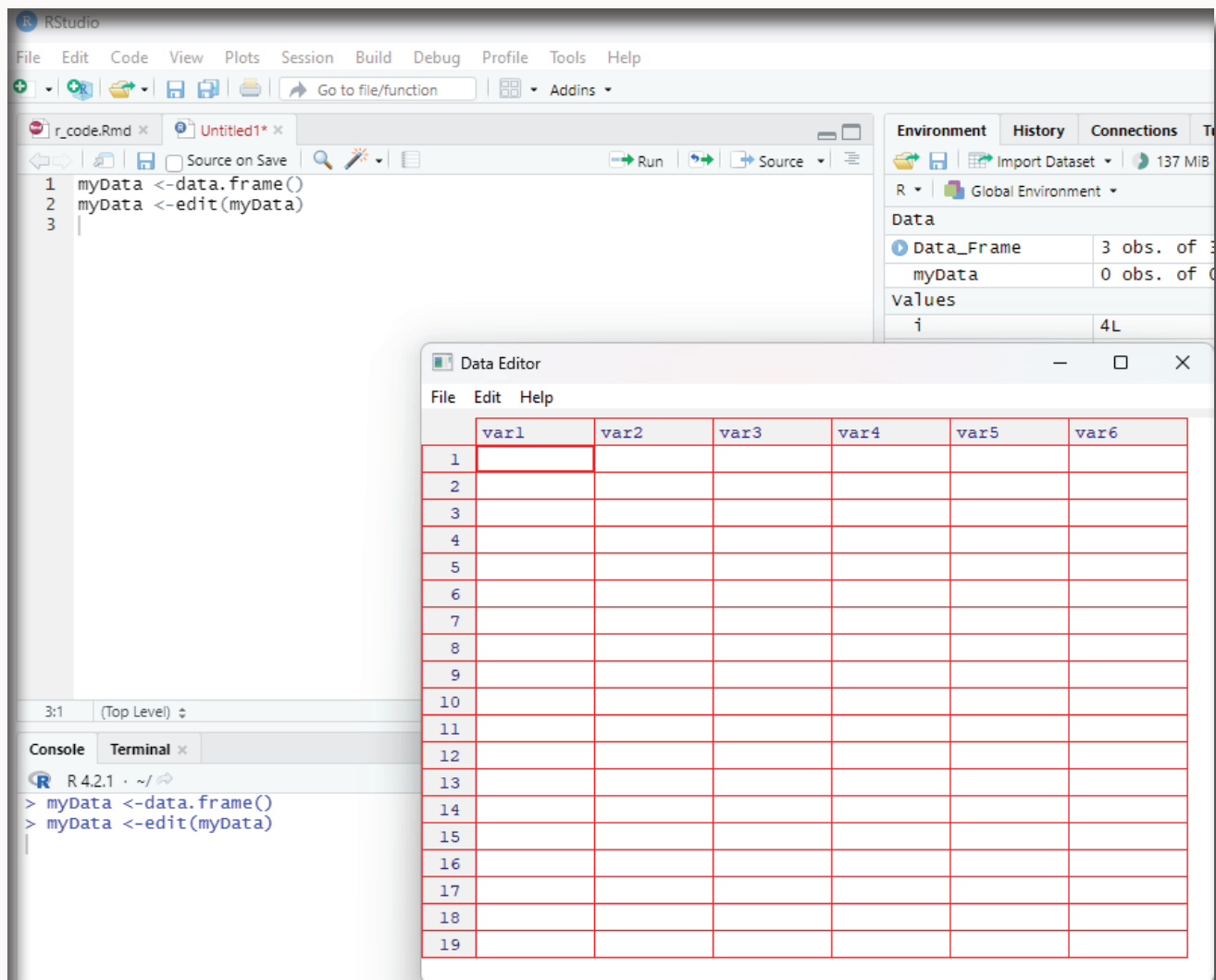


Image showing data editor launched

Example:

To create a sample data frame with 5 columns:

data_entry <-data.frame(One=1:5, Two=6:10, Three=11:15)

data_entry

To list out the column names the following code is to be used.

names(data_entry)

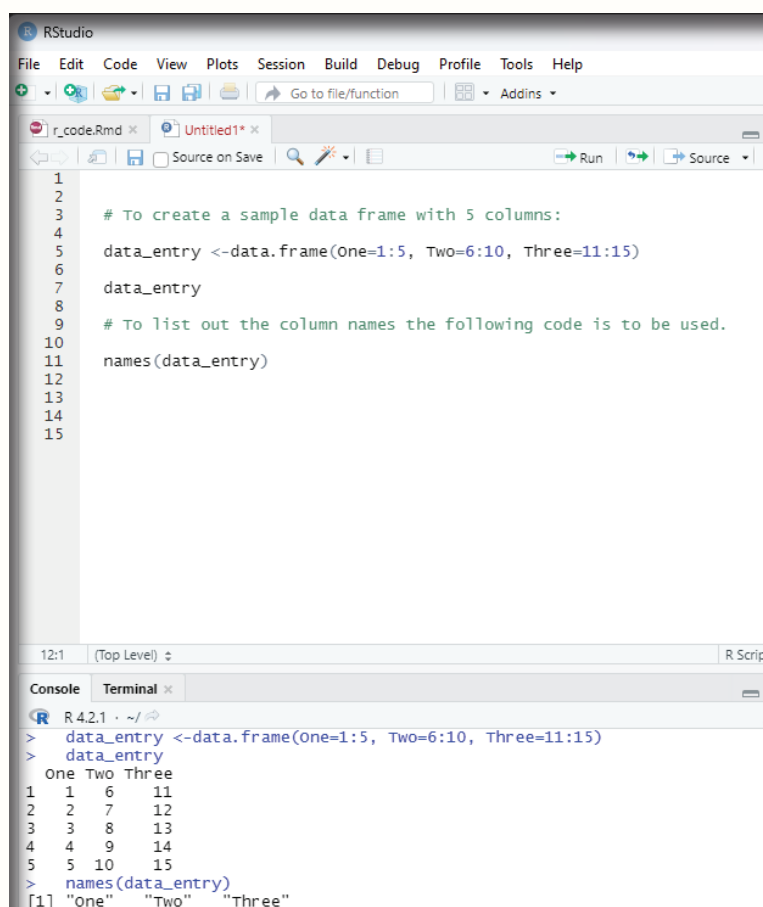
Renaming the column names.

Library plyr should be loaded first.

library(plyr)

rename(data_entry, c("One" = "1", "Two" = "2", "Three"="3"))

Entering data into RStudio is a bit tricky for a beginner. The best way is to import data created from other data base software like Excel, SPSS etc which provide a convenient way of data entry because of their default column and row features. Imported data can be subjected to analysis within R environment. Data can be imported using the File menu - under which import data set is listed. The user can choose the data format to import data from. RStudio if needed will seek to download some libraries for seamless import of data set created in other software if connected to Internet.



The screenshot shows the RStudio interface. The script editor contains the following R code:

```
1  
2  
3 # To create a sample data frame with 5 columns:  
4 data_entry <- data.frame(One=1:5, Two=6:10, Three=11:15)  
5  
6 data_entry  
7  
8  
9 # To list out the column names the following code is to be used.  
10 names(data_entry)  
11  
12  
13  
14  
15
```

The console output shows the execution of the code:

```
> data_entry <- data.frame(One=1:5, Two=6:10, Three=11:15)  
> data_entry  
  One Two Three  
1   1   6   11  
2   2   7   12  
3   3   8   13  
4   4   9   14  
5   5  10   15  
> names(data_entry)  
[1] "One" "Two" "Three"
```

Image showing names of columns listed

All along in this book the user would have been exposed to the code that creates “vectors” of data.

Example:

```
variable_1=c(1,2,3,4,5)
```

If the user prefers entering data in a spread sheet window R needs to be convinced to present the interface by using a code shown below:

```
data.entry(1)
```

This command opens up a data editor window with a column named 1 and the row is also named 1. This can easily be edited by clicking on the value. Up and down arrows can be used to navigate the worksheet. When data entry is complete then the user can choose file>close. This closes the data editor after saving its contents.

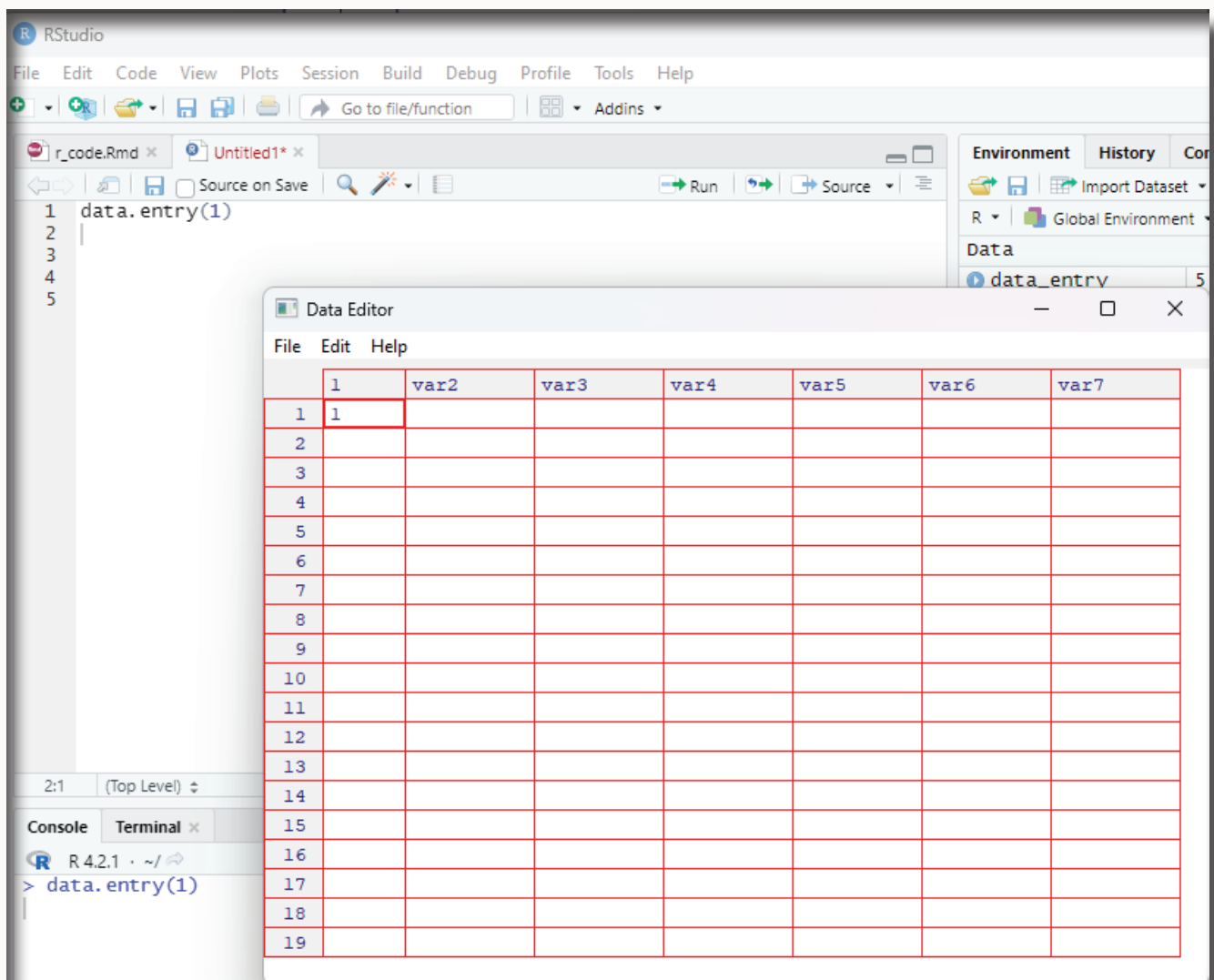


Image showing data entry window opening up after keying `data.entry(1)`

The data entry window should be closed before entering new commands in the R console. Using the console window data values can be changed using the following command:

```
data.entry(variablename)
```

The user can list any number of variables separated by a comma within the bracket.

The user can also open a dialog box to import data stored in csv format (comma separated values). Excel files can also be stored as .csv files.

The user can also open a dialog window to find the data file that needs to be imported into R.

```
testdata = read.table(path to the file that needs to be imported).
```

Taking input from user in R Programming.

This is an important feature in R. Like all programming languages in R also it is possible to take input from the user. This is an important aspect in data collection. This is made possible by using:

readline() method

scan() method

Using readline() method:

In this method R takes input in a string format. If the user inputs an integer then it is inputted as a string. If the user wants to input 320, then it will input as "320" like a string. The user hence will have to convert the inputted value into the format that is needed for data analysis. In the above example the string "320" will have to be converted to integer 320. In order to convert the inputted value to the desired data type, the following functions can be used.

as.integer(n) - convert to integer

as.numeric(n) - convert to numeric (float, double etc)

as.complex(n) - convert to complex number (3+2i)

as.Date(n) - convert to date etc.

Syntax:

```
var =readline();  
var=as.integer(var);
```

Example:

```
# Taking input from the user  
# This is done using readline() method  
# This command will prompt the user to input the desired value  
  
var = readline();  
  
# Convert the inputted value to integer
```

```
var= as.integer(var);
```

```
# Print the value
```

```
print(var)
```

One can also show a message in the console window to inform the user, what to input the program with. This can be done using an argument named prompt inside the readline() function.

Syntax:

```
var1 = readline(prompt="Enter any number:");
```

or

```
var1 = readline("Enter any Number:");
```

Code:

```
# Taking input with showing the message.
```

```
var = readline(prompt = "Enter any number : ");
```

```
# Convert the inputted value to an integer.
```

```
var = as.integer(var);
```

```
# Print the value
```

```
print(var)
```

Taking multiple inputs in T:

This action is similar to that of taking a single output, but it just needs multiple readline() inputs. One can use braces to define multiple readline() inside it.

Syntax:

```
var1=readline("Enter 1st number:");
```

```
var2=readline("Enter 2nd number:");
```

```
var3=readline("Enter 3rd number:");
```

or

```
# Taking multiple inputs from the user
```

```
{  
var1 = readline("Enter 1st number : ");  
var2 = readline("Enter 2nd number : ");  
var3 = readline("Enter 3rd number : ");  
var4 = readline("Enter 4th number : ");  
}
```

```
# Converting each value to integer
```

```
var1 = as.integer(var1);  
var2 = as.integer(var2);  
var3 = as.integer(var3);  
var4 = as.integer(var4);
```

```
# print the sum of the 4 numbers
```

```
print(var1+var2+var3+var4)
```

```
# R program to illustrate taking input from the user
```

```
# string input
```

```
var1 = readline(prompt = "Enter your name : ");
```

```
# character input
```

```
var2 = readline(prompt = "Enter any character : ");
```

```
# convert to character
```

```
var2 = as.character(var2)
```

```
# printing values
```

```
print(var1)
```

```
print(var2)
```

```
1  
2 {  
3   var1 = readline("Enter 1st number : ");  
4   var2 = readline("Enter 2nd number : ");  
5   var3 = readline("Enter 3rd number : ");  
6   var4 = readline("Enter 4th number : ");  
7 }  
8 # converting each value  
9 var1 = as.integer(var1);  
10 var2 = as.integer(var2);  
11 var3 = as.integer(var3);  
12 var4 = as.integer(var4);  
13  
14 # print the sum of the 4 number  
15 print(var1 + var2 + var3 + var4)  
16
```

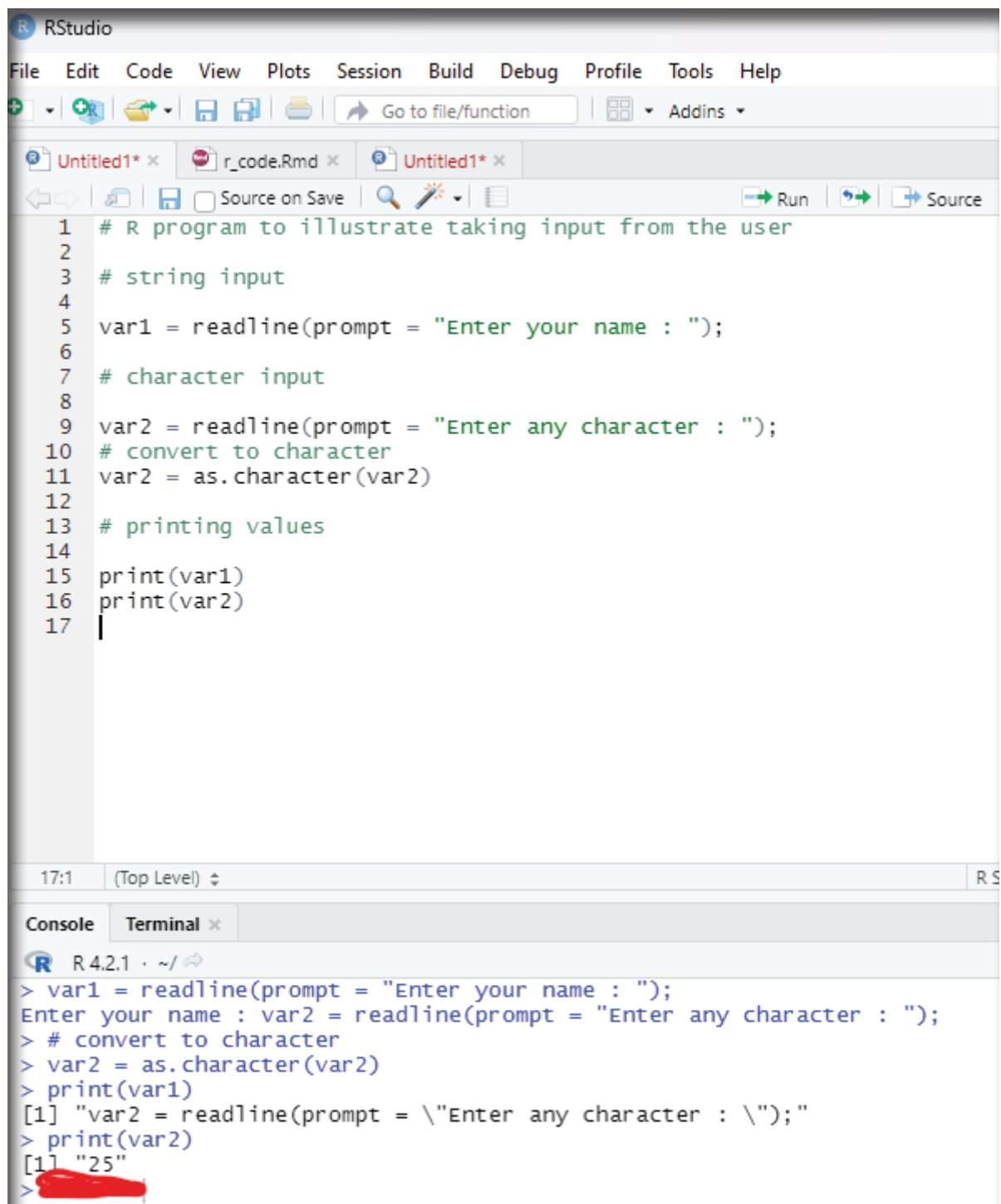
2:1 (Top Level) ↕

Console Terminal ×

R 4.2.1 · ~/

```
+ var1 = readline("Enter 1st number : ");  
+ var2 = readline("Enter 2nd number : ");  
+ var3 = readline("Enter 3rd number : ");  
+ var4 = readline("Enter 4th number : ");  
+ }  
Enter 1st number : 2  
Enter 2nd number : 3  
Enter 3rd number : 22  
Enter 4th number : 76  
> # converting each value  
> var1 = as.integer(var1);  
> var2 = as.integer(var2);  
> var3 = as.integer(var3);  
> var4 = as.integer(var4);  
> # print the sum of the 4 number  
> print(var1 + var2 + var3 + var4)  
[1] 103
```

Image showing readline and converting as integer functions



The image shows the RStudio interface. The top menu bar includes File, Edit, Code, View, Plots, Session, Build, Debug, Profile, Tools, and Help. Below the menu is a toolbar with icons for file operations and a search bar. The editor pane shows a script with the following code:

```
1 # R program to illustrate taking input from the user
2
3 # string input
4
5 var1 = readline(prompt = "Enter your name : ");
6
7 # character input
8
9 var2 = readline(prompt = "Enter any character : ");
10 # convert to character
11 var2 = as.character(var2)
12
13 # printing values
14
15 print(var1)
16 print(var2)
17
```

The status bar at the bottom of the editor shows '17:1 (Top Level)'. Below the editor is a console pane with the following output:

```
R 4.2.1 ~/> var1 = readline(prompt = "Enter your name : ");
Enter your name : var2 = readline(prompt = "Enter any character : ");
> # convert to character
> var2 = as.character(var2)
> print(var1)
[1] "var2 = readline(prompt = \"Enter any character : \");"
> print(var2)
[1] "25"
>
```

The last line of the console output, ">", is highlighted with a red brush.

Image showing string input

Using scan() method:

Another method to take user input in R language is to use a method known as scan() method. This method takes input from the console. This is rather handy when inputs are needed to be taken quickly for any mathematical calculation or for any dataset. This method reads data in the form of a vector or list. This method can also be used to read input from a file also.

Syntax:

```
x=scan()
```

scan() method is taking input continuously. In order to terminate the input process one needs to press ENTER key 2 times on the console.

Example:

This code is to take input using scan() method, where some integer number is taken as input and the same value is printed in the next line of the console.

```
# R program to illustrate  
# taking input from the user  
  
# taking input using scan()  
x = scan()  
# print the inputted values  
print(x)
```

Scan function is used for scanning text files.

Example:

```
# create a data frame  
  
data <- data.frame(x1 = c(4, 4, 1, 9),  
                  x2 = c(1, 8, 4, 0),  
                  x3 = c(5, 3, 5, 6))  
data  
  
#write data as text file to directory  
  
write.table(data,  
            file = "data.txt",  
            row.names = FALSE)  
  
data1 <- scan("data.txt"), what = "character")
```

```
data1
```

```
data1 <- scan("data.txt", what = "character")
```

```
data1
```

This code has created a vector, that contains all values of the data in the data frame including the column names.

Scan command can also be used to read data into a list. The code below creates a list with three elements. Each of the list elements contains one column of the original data frame. The data in the data file is scanned line by line.

```
# Read txt file into list
```

```
data2 <- scan("data.txt", what = list("", "", ""))
```

```
# Print scan output to the console
```

```
data2
```

Skipping lines with scan function:

Scan function provides additional specifications. One of which is the skip function. This option allows the user to skip any line of the input file. Since the column names are usually the first input lines of a file, one can skip them with the specification skip = 1.

```
# Skip first line of txt file
```

```
data3 <- scan("data.txt", skip = 1)
```

```
# Print scan output to the console
```

```
data3
```

Scanning Excel CSV file:

Scan function can also be used to scan CSV file created by Excel.

Example:

```
write.table(data,  
  file = "data.csv",  
  row.names = FALSE)
```

```
data4 <-scan("data.csv", what = "character")
```

data4

Scanning RStudio console:

Another functionality of scan is that it can be used to read input from the RStudio console.

Example:

```
data5 <-scan("")
```

read function can also be used in lieu of scan function.

```
read.csv  
read.table  
readLines  
n.readLines
```

read.csv function:

In order to import csv files the import function available under file menu of RStudio can be used.

As a first step using a notepad the user can create a small data base with details as shown below:

```
id,name,salary,start_date,dept  
1,simon,623.3,2012-01-01,IT  
2,Ashok,515.2,2013-09-23,Operations  
3,Adil,611,2014-11-15,IT  
4,Murray,729,2014-05-11,HR  
5,Sunil,843.25,2015-03-27,Finance  
6,Naina,578,2013-05-21,IT  
7,Seetha,632.8,2013-07-30,Operations  
8,Gautham,722.5,2014-06-17,Finance
```

Every column should be separated by a comma that is the reason why it is called as comma separated values (.CSV).

Import Dataset

Name:

Encoding:

Heading: ☒ Yes ☐ No

Row names:

Separator:

Decimal:

Quote:

Comment:

na.strings:

☐ Strings as factors

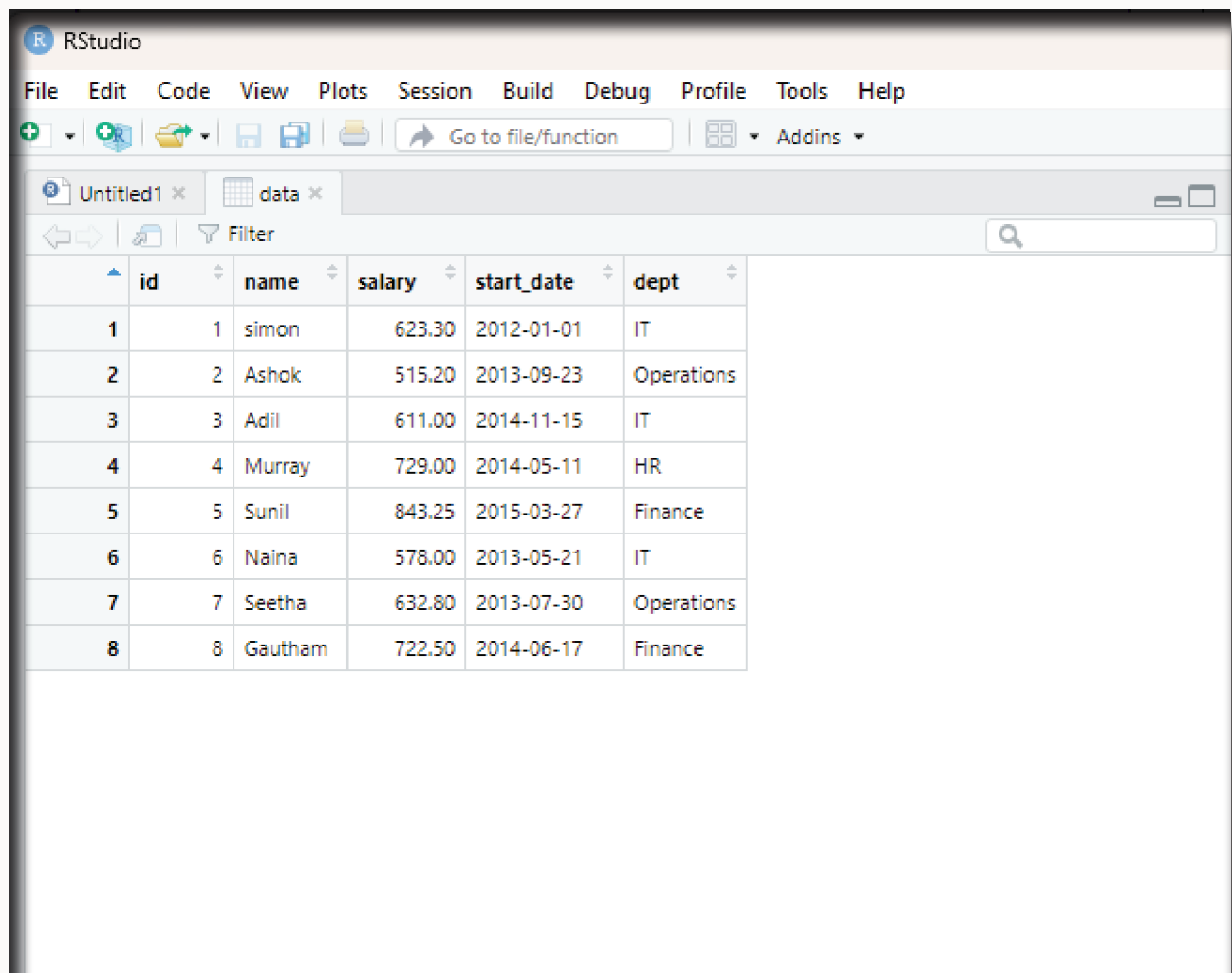
Input File

```
id,name,salary,start_date,dept
1,simon,623.3,2012-01-01,IT
2,Ashok,515.2,2013-09-23,Operations
3,Adil,611,2014-11-15,IT
4,Murray,729,2014-05-11,HR
5,Sunil,843.25,2015-03-27,Finance
6,Naina,578,2013-05-21,IT
7,Seetha,632.8,2013-07-30,Operations
8,Gautham,722.5,2014-06-17,Finance
```

Data Frame

id	name	salary	start_date	dept
1	simon	623.30	2012-01-01	IT
2	Ashok	515.20	2013-09-23	Operations
3	Adil	611.00	2014-11-15	IT
4	Murray	729.00	2014-05-11	HR
5	Sunil	843.25	2015-03-27	Finance
6	Naina	578.00	2013-05-21	IT
7	Seetha	632.80	2013-07-30	Operations
8	Gautham	722.50	2014-06-17	Finance

Image showing the import screen. Before clicking on the import button the user should verify if all the settings are given as shown in the screenshot



The screenshot shows the RStudio interface with the 'data' viewer displaying a table of employee data. The table has 8 rows and 6 columns: id, name, salary, start_date, and dept. The data is as follows:

	id	name	salary	start_date	dept
1	1	simon	623.30	2012-01-01	IT
2	2	Ashok	515.20	2013-09-23	Operations
3	3	Adil	611.00	2014-11-15	IT
4	4	Murray	729.00	2014-05-11	HR
5	5	Sunil	843.25	2015-03-27	Finance
6	6	Naina	578.00	2013-05-21	IT
7	7	Seetha	632.80	2013-07-30	Operations
8	8	Gautham	722.50	2014-06-17	Finance

Image showing the imported data displayed in the RStudio scripting window

Analyzing the data imported:

Command to analyze the imported data.

```
print (is.data.frame(data))
```

```
print (ncol(data))
```

```
print (nrow(data))
```

Output:

```
print (is.data.frame(data))  
[1] TRUE  
> print (ncol(data))  
[1] 5  
> print (nrow(data))  
[1] 8
```

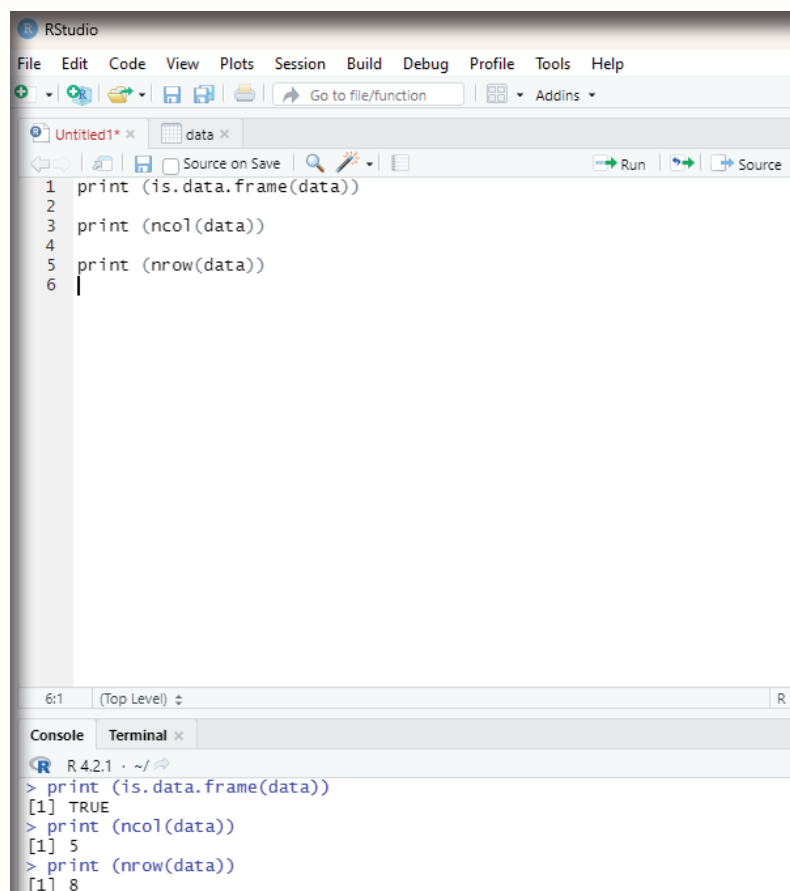


Image showing the output when data analysis is performed

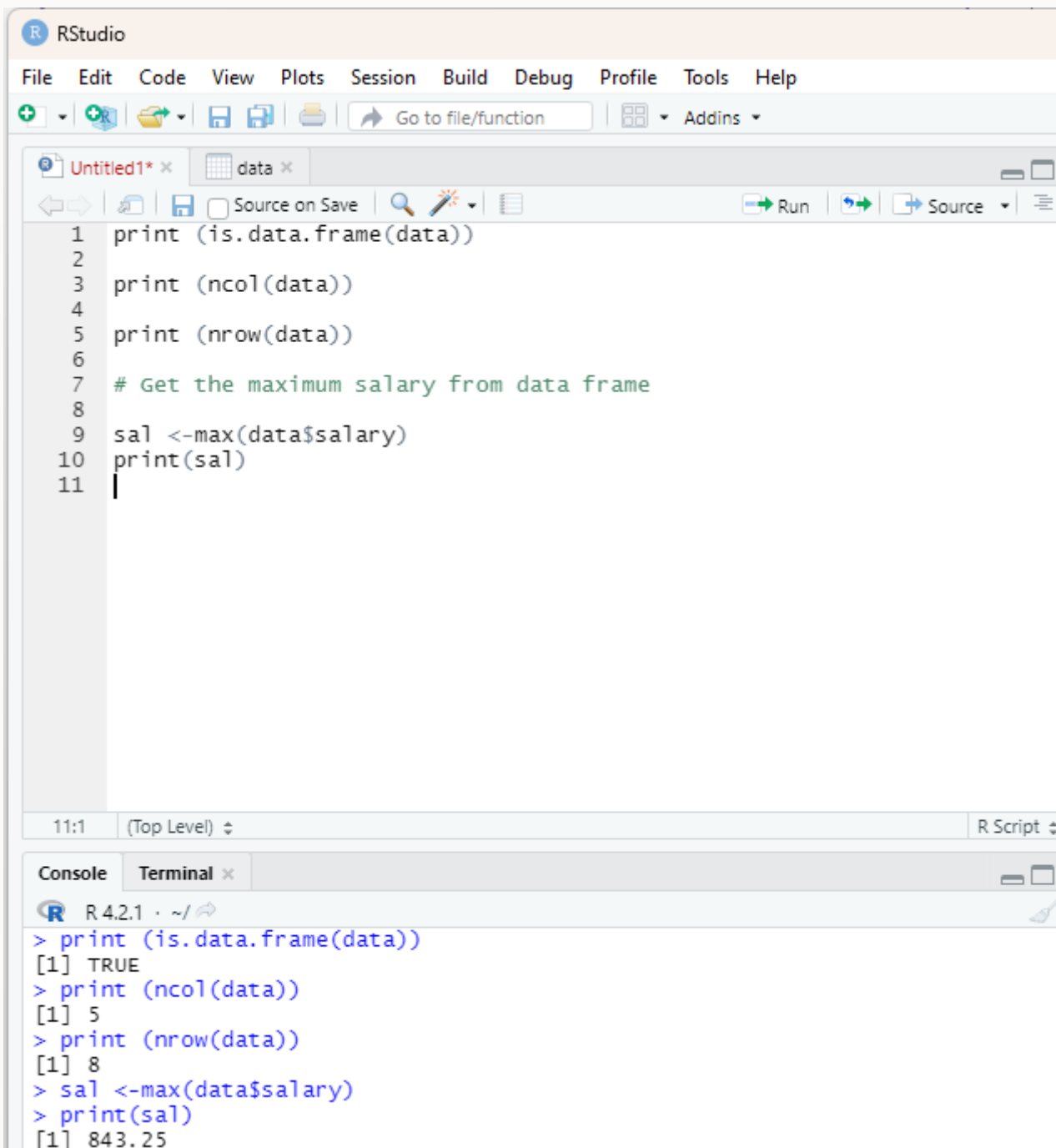
From the sample data the user is encouraged to get the maximum salary of the employee by using R code.

Get the maximum salary from data frame

```
sal <-max(data$salary)  
print(sal)
```

Output:

843.25.



The screenshot shows the RStudio interface. The top menu bar includes File, Edit, Code, View, Plots, Session, Build, Debug, Profile, Tools, and Help. Below the menu is a toolbar with icons for file operations and a search bar. The main editor window displays a script with the following code:

```
1 print (is.data.frame(data))  
2  
3 print (ncol(data))  
4  
5 print (nrow(data))  
6  
7 # Get the maximum salary from data frame  
8  
9 sal <-max(data$salary)  
10 print(sal)  
11 |
```

The status bar at the bottom of the editor shows '11:1 (Top Level) R Script'. Below the editor is a console window with the following output:

```
R 4.2.1 ~/  
> print (is.data.frame(data))  
[1] TRUE  
> print (ncol(data))  
[1] 5  
> print (nrow(data))  
[1] 8  
> sal <-max(data$salary)  
> print(sal)  
[1] 843.25
```

Image showing the maximum salary displayed

In order to get the details of person drawing maximum salary the code used is:

Retrieve value of the person detail having maximum salary.

```
retval <- subset(data, salary == max(salary))  
print(retval)
```

To get details of all persons working in IT department:

```
retval <- subset(data, dept == "IT")  
print(retval)
```

To get persons in IT department whose salary is greater than 600.

```
info <- subset(data, salary > 600 & dept == "IT")  
print(info)
```

Given below is the entire sequential code for all the functions detailed above:

```
print(is.data.frame(data))  
print(ncol(data))  
print(nrow(data))
```

```
# Get the max salary from data frame.  
sal <- max(data$salary)  
print(sal)
```

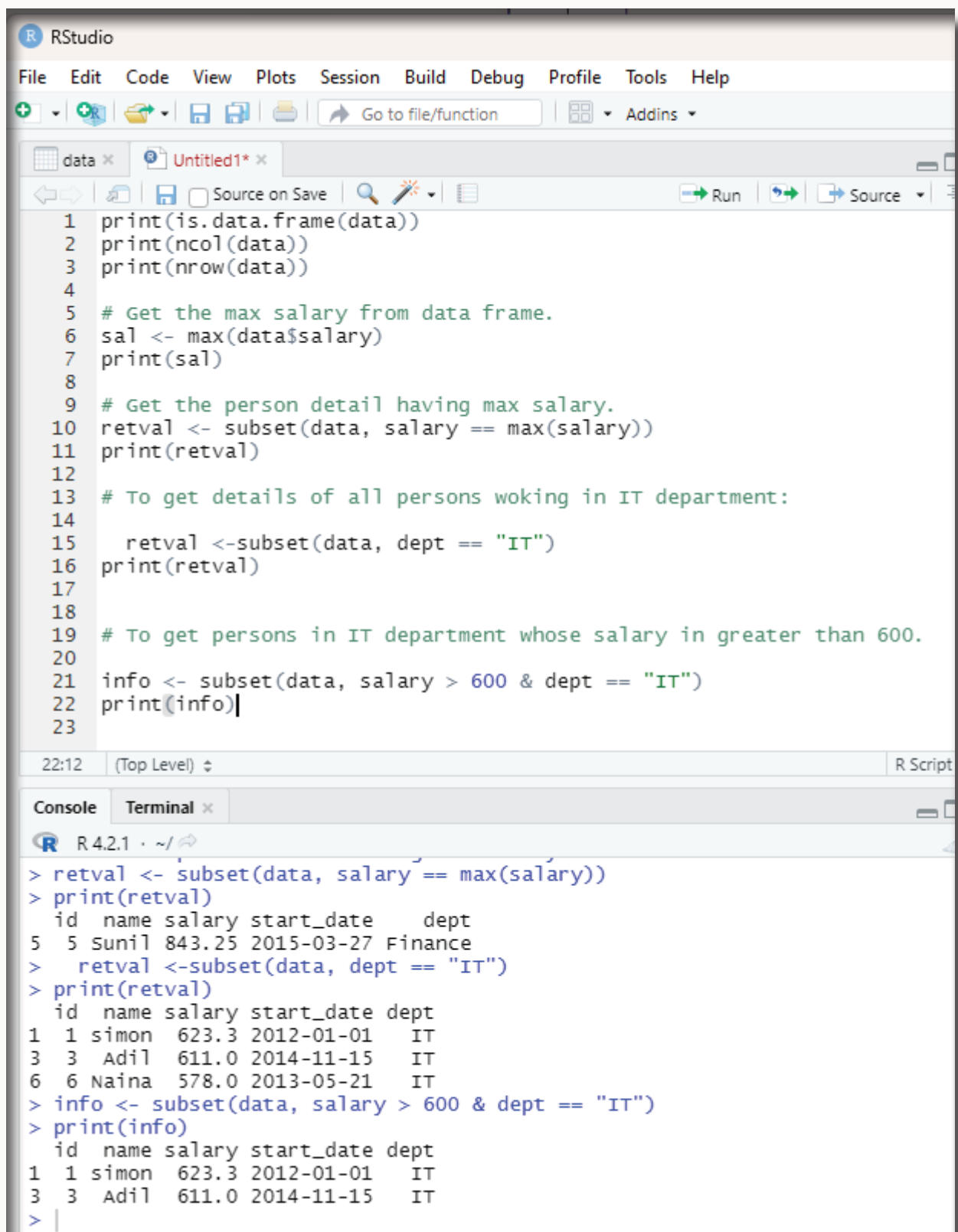
```
# Get the person detail having max salary.  
retval <- subset(data, salary == max(salary))  
print(retval)
```

```
# To get details of all persons working in IT department:
```

```
retval <- subset(data, dept == "IT")  
print(retval)
```

```
# To get persons in IT department whose salary is greater than 600.
```

```
info <- subset(data, salary > 600 & dept == "IT")  
print(info)
```



The image shows the RStudio interface with a script editor and a console. The script editor contains the following R code:

```
1 print(is.data.frame(data))
2 print(ncol(data))
3 print(nrow(data))
4
5 # Get the max salary from data frame.
6 sal <- max(data$salary)
7 print(sal)
8
9 # Get the person detail having max salary.
10 retval <- subset(data, salary == max(salary))
11 print(retval)
12
13 # To get details of all persons working in IT department:
14
15 retval <- subset(data, dept == "IT")
16 print(retval)
17
18
19 # To get persons in IT department whose salary is greater than 600.
20
21 info <- subset(data, salary > 600 & dept == "IT")
22 print(info)
23
```

The console shows the output of the code:

```
> retval <- subset(data, salary == max(salary))
> print(retval)
  id name salary start_date dept
5  5 Sunil 843.25 2015-03-27 Finance
> retval <- subset(data, dept == "IT")
> print(retval)
  id name salary start_date dept
1  1 Simon  623.3 2012-01-01   IT
3  3 Adil   611.0 2014-11-15   IT
6  6 Naina  578.0 2013-05-21   IT
> info <- subset(data, salary > 600 & dept == "IT")
> print(info)
  id name salary start_date dept
1  1 Simon  623.3 2012-01-01   IT
3  3 Adil   611.0 2014-11-15   IT
> |
```

Image showing the sequential code for all the functions described above

Importing data directly from Excel:

Excel is the most commonly used data base soft ware. In order to import data directly from Excel certain libraries need to b installed in RStudio.

These packages include:

XLConnect

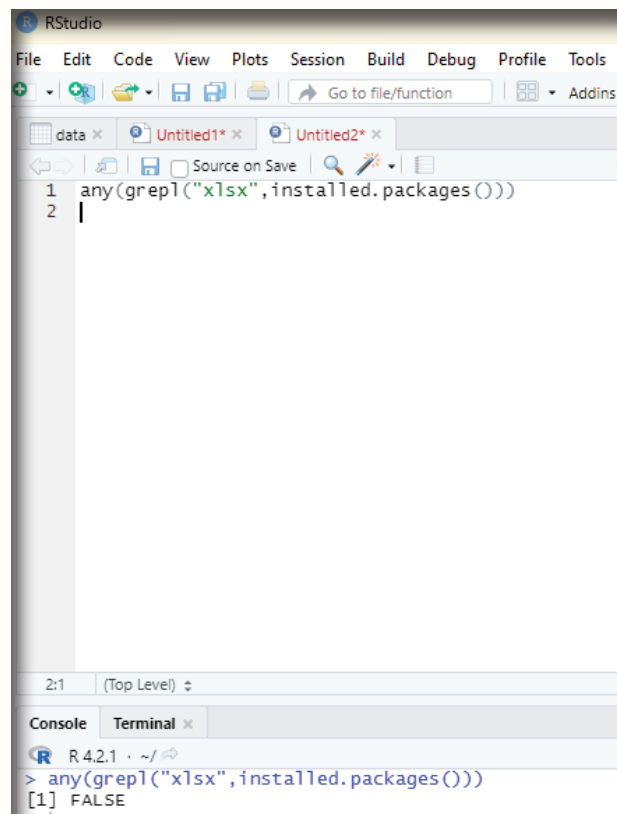
xlsx

gdata

xlsx can be installed via package manager. Before that the user can verify whether the package is available within R environment by using the code:

```
any(grepl("xlsx",installed.packages()))
```

If the output displays the value TRUE it is installed. If FALSE is displayed then the package should be installed by the user.



The screenshot shows the RStudio interface. The top menu bar includes File, Edit, Code, View, Plots, Session, Build, Debug, Profile, and Tools. Below the menu is a toolbar with icons for file operations and a search bar. The main editor window displays two untitled files. The first file contains the R code:

```
1 any(grepl("xlsx", installed.packages()))
2 |
```

 The bottom panel shows the Console window with the command

```
> any(grepl("xlsx", installed.packages()))
```

 and the output

```
[1] FALSE
```

.

Image showing that xlsx installation not available in the R package.

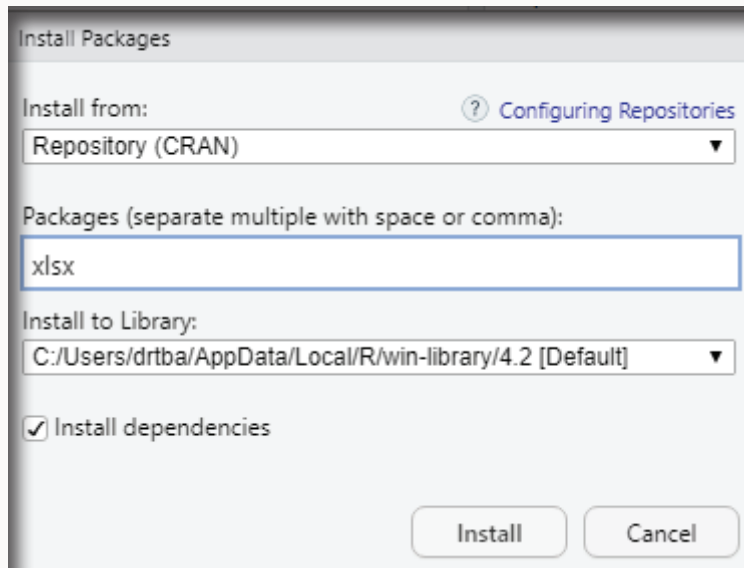


Image showing xlsx package being installed using package install manager

xlsx package that has been installed should be enabled from the packages window found in the left bottom area of RStudio.

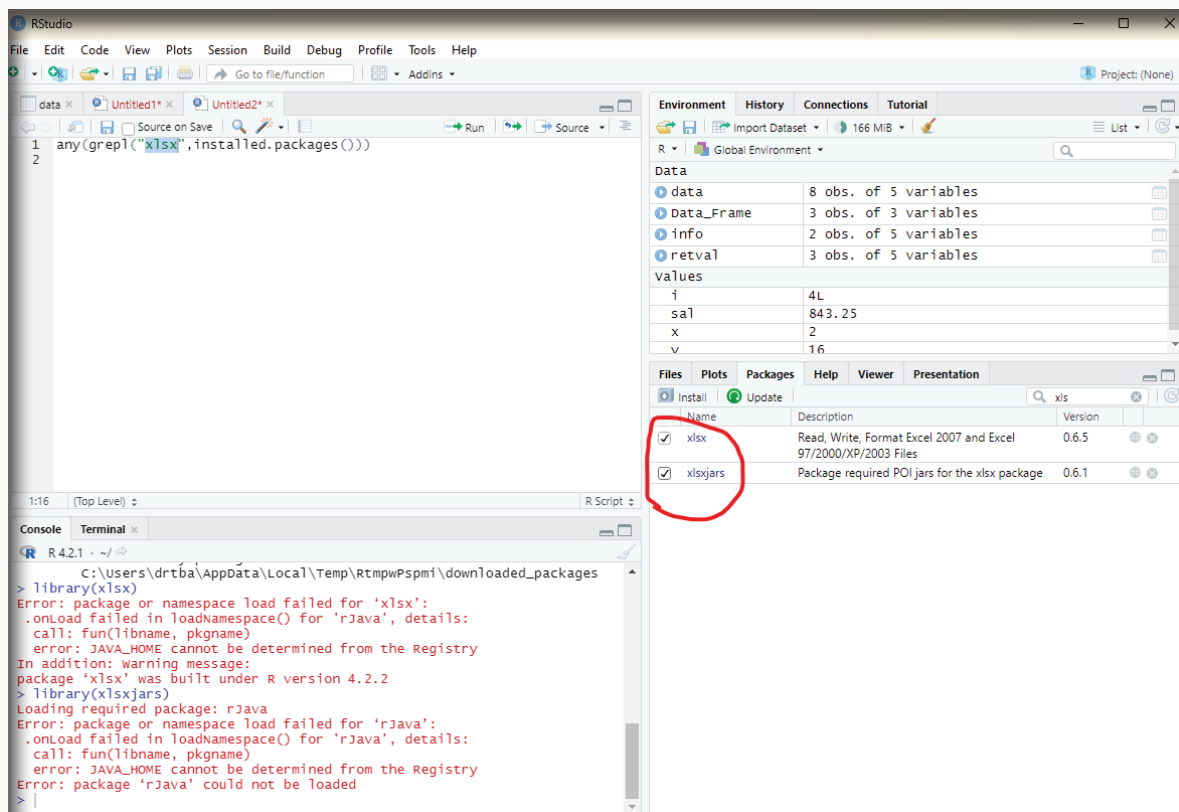


Image showing xlsx package enabled

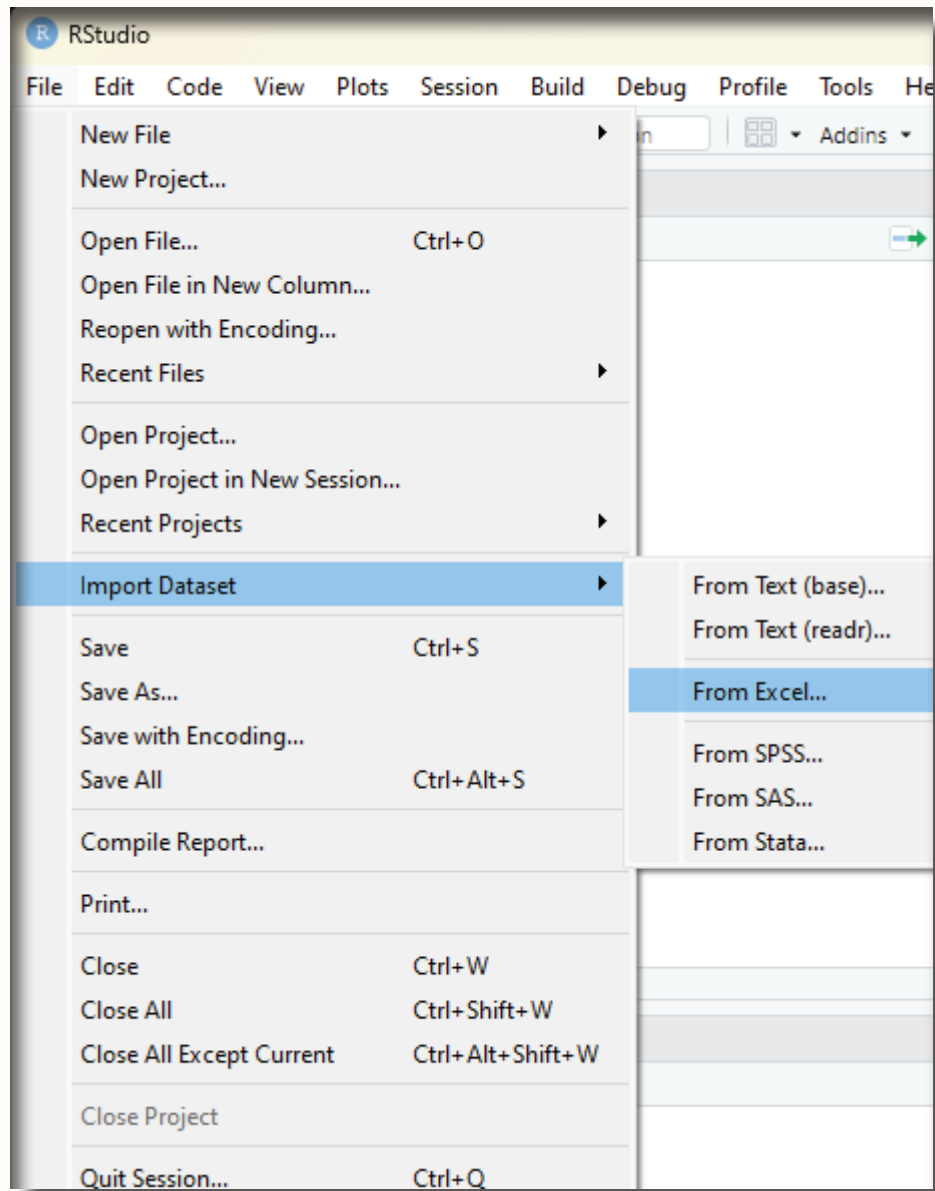


Image showing excel data set being imported to RStudio using the sub menu listed under import dataset.

Import Excel Data

File/URL:
~/marks.xlsx Browse...

Data Preview:

Name (character)	Maths (double)	Physics (double)	Chemistry (double)
Arjun	88	67	87
Ajith	92	45	89
Aruna	89	78	88
Ajitha	100	92	97
Ajith	78	68	79
Babu	87	76	88
Chandar	56	66	67

Previewing first 50 entries.

Import Options:

Name: Max Rows:
 Sheet: Skip:
 Range: NA:

☒ First Row as Names
☒ Open Data Viewer

Code Preview:

```
library(readxl)
marks <- read_excel("marks.xlsx")
view(marks)
```

? Reading Excel files using readxl Import Cancel

Image showing Excel data set import screen

Read the first worksheet in the file input.xlsx.

```
data <- read.xlsx("marks.xlsx", sheetIndex = 1)
print(data)
```

Data Analysis in R Programming

First step in data analysis is to load the data in to R interface. This can be done by directly entering data directly into R using Data editor interface. Data from other data software like Excel can be directly imported into R.

Use of data string function

```
str(data_name)
```

This function helps in understanding the structure of data set, data type of each attribute and number of rows and columns present in the data.

In order to learn to analyze data using R programming titanic data base can be installed into R environment to facilitate learning the nuts and bolts of data analysis.

Code to install titanic data base.

In the scripting window the following code should be keyed and made to run.

```
install.packages("titanic")
```

After installation the package “titanic” should be initialized by selecting the box in front of titanic package name in the package window.

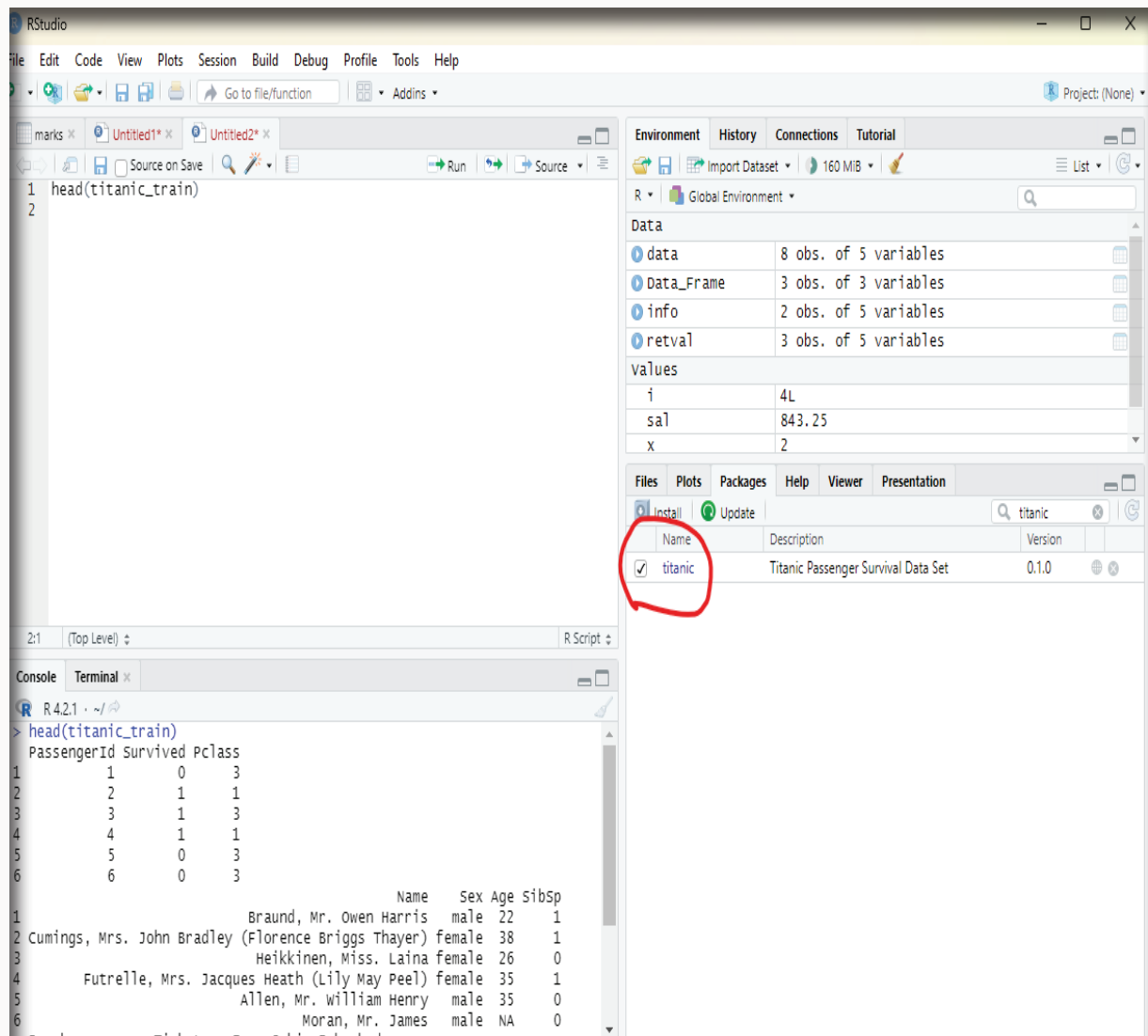


Image showing titanic data base enabled

The first step in data analysis is basic exploration to see the data. Head and tail function is used to see how the data looks like. The head function reveals to the user the first six rows of the data and the tail function reveals the last six items. This will enable the user to spot the field of interest in the data set that is subjected to the study.

head(titanic_train)

tail(titanic_train)

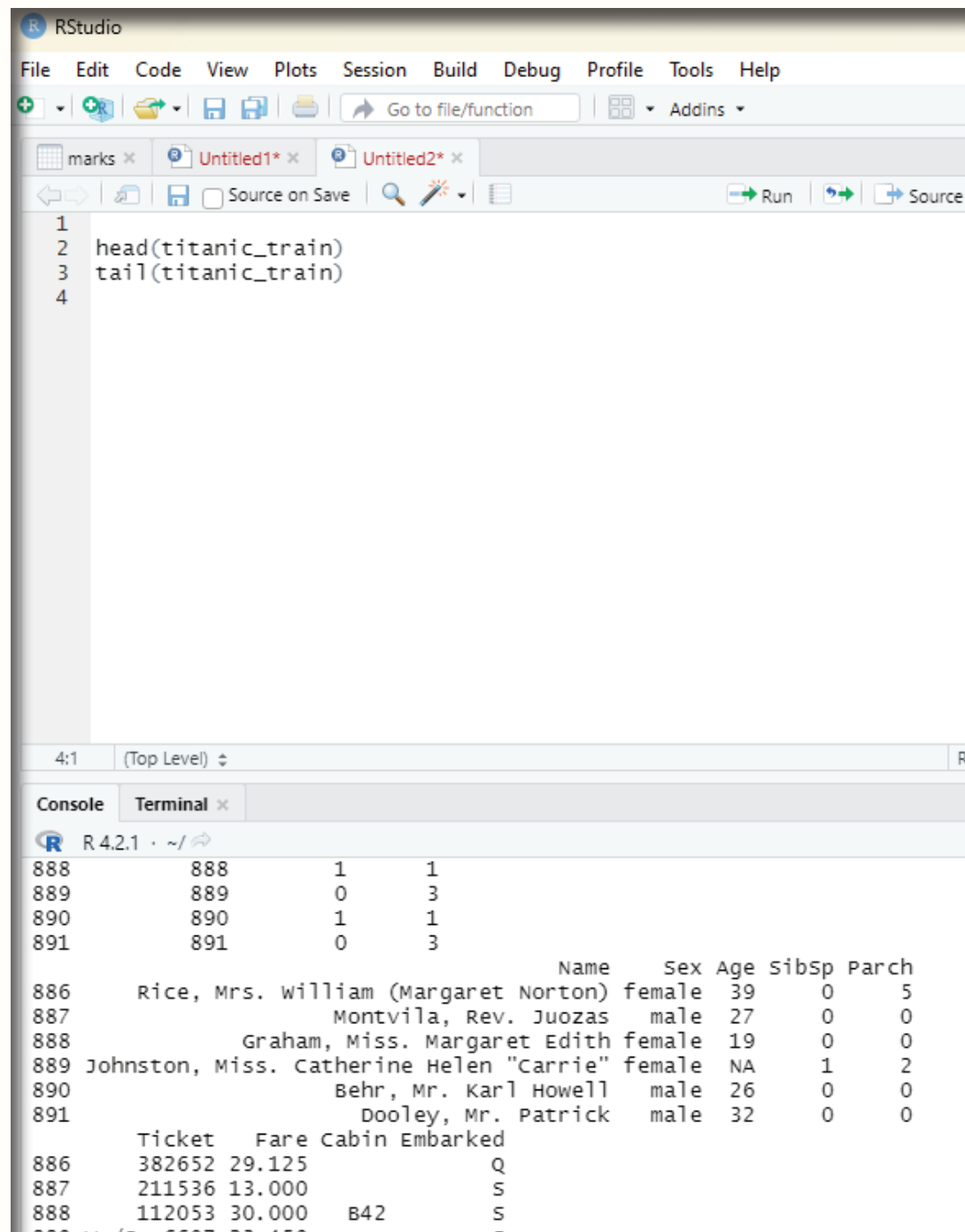


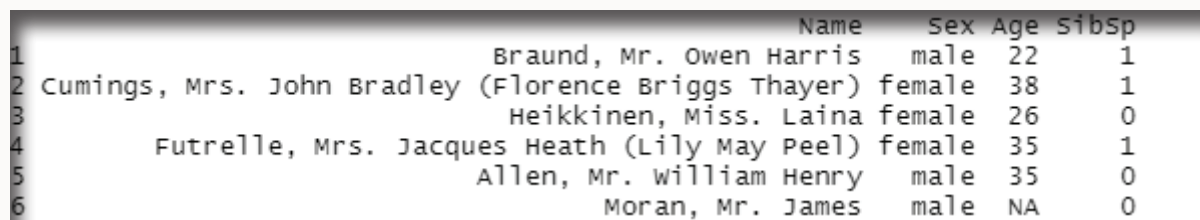
Image showing data from data set "titanic" revealing the first and last 6 items

The basic exploration of the data set reveals the following interesting data:

Sex

Age

SibSp (Number of Siblings/Spouses Abroad)



	Name	Sex	Age	SibSp
1	Braund, Mr. Owen Harris	male	22	1
2	Cumings, Mrs. John Bradley (Florence Briggs Thayer)	female	38	1
3	Heikkinen, Miss. Laina	female	26	0
4	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35	1
5	Allen, Mr. William Henry	male	35	0
6	Moran, Mr. James	male	NA	0

Image showing the interesting data columns as revealed by the head and tail command

Summary of the data base containing the minimum values, maximum values, median, mode, first, second and third quartiles etc.

Code:

```
summary(titanic_train)
```

```

Class :character 1st Qu.:20.12 1st Qu.:0.000 1st Qu.:0.0000
Mode :character Median :28.00 Median :0.000 Median :0.0000
                Mean  :29.70 Mean  :0.523 Mean  :0.3816
                3rd Qu.:38.00 3rd Qu.:1.000 3rd Qu.:0.0000
                Max.   :80.00 Max.   :8.000 Max.   :6.0000
                NA's   :177
Ticket          Fare          Cabin
Length:891      Min.   : 0.00 Length:891
Class :character 1st Qu.: 7.91 Class :character
Mode :character Median :14.45 Mode :character
                Mean  :32.20
                3rd Qu.:31.00
                Max.   :512.33

```

Image showing the display of summary of the data set

The class of each column can be studied using the apply function.

```
sapply(titanic_train, class)
```

This will help the user to identify the type of data in a particular column.

```

> sapply(titanic_train, class)
PassengerId  Survived    Pclass      Name      Sex      Age
"integer"    "integer"  "integer" "character" "character" "numeric"
 Sibsp      Parch      Ticket    Fare      Cabin    Embarked
"integer"    "integer" "character" "numeric" "character" "character"

```

Image showing output from sapply function

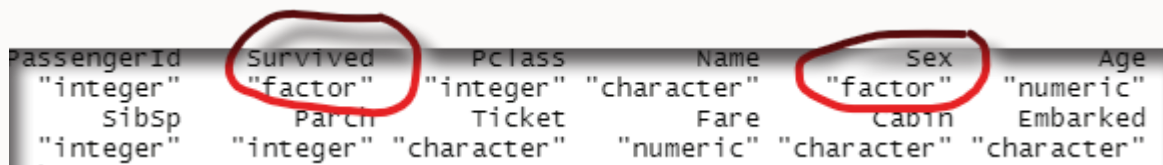
This function is rather important because data summarization could be inaccurate if different classes of data are compared

From the above summary it can be observed that data under “Survived” belongs to the class integer and data under “Sex” is character. In order to run a good summary, the classes need to be changed.

```
titanic_train$Survived = as.factor(titanic_train$Survived)
```

```
titanic_train$Sex = as.factor(titanic_train$Sex)
```

This command will change the class of the column “survived” and “Sex” into factors that will also change the way in which data is summarized.



PassengerId	Survived	Pclass	Name	Sex	Age
"integer"	"factor"	"integer"	"character"	"factor"	"numeric"
SibSp	Parch	Ticket	Fare	Cabin	Embarked
"integer"	"integer"	"character"	"numeric"	"character"	"character"

Image showing the results after conversion of data inside the columns “Survived” and “Sex” into factors

Preparing the data:

Before performing any other task on the data set the user should perform one important check. It is to ascertain if there are any missing data. This can be performed using the following code:

```
is.na(titanic_train)
```

```
sum(is.na(titanic_train))
```

is.na will check if the data is NA or not and return the result as true or false. One can also use sum(is.na(#object)) to count how many NA data there are.

The screenshot shows the RStudio interface with the following components:

- Menu Bar:** File, Edit, Code, View, Plots, Session, Build, Debug, Profile, Tools, Help.
- Toolbar:** Includes icons for file operations (new, open, save, print), a search bar labeled "Go to file/function", and an "Addins" dropdown.
- Source Editor:** Contains the following R code:

```
1  
2 head(titanic_train)  
3 tail(titanic_train)  
4 summary(titanic_train)  
5 sapply(titanic_train, class)  
6 titanic_train$Survived = as.factor(titanic_train$Survived)  
7  
8 titanic_train$sex = as.factor(titanic_train$sex)  
9 sapply(titanic_train, class)  
10  
11 is.na(titanic_train)  
12 sum(is.na(titanic_train))  
13  
14
```
- Console:** Shows the output of the code execution:

```
R 4.2.1 ~/  
[73,] FALSE FALSE FALSE  
[74,] FALSE FALSE FALSE  
[75,] FALSE FALSE FALSE  
[76,] FALSE FALSE FALSE  
[77,] FALSE FALSE FALSE  
[78,] FALSE FALSE FALSE  
[79,] FALSE FALSE FALSE  
[80,] FALSE FALSE FALSE  
[81,] FALSE FALSE FALSE  
[82,] FALSE FALSE FALSE  
[83,] FALSE FALSE FALSE  
[ reached getOption("max.print") -- omitted 808 rows ]  
> sum(is.na(titanic_train))
```

Image showing the results containing NA in the data set

Since missing data might disturb some analysis, it is better if they could be excluded. Ideally the entire row that has the missing data should be excluded.

```
titanic_train_droppedna = titanic_train[rowSums(is.na(titanic_train)) <=0,]
```

This script will dropout any row that has missing data on it. Using this method the user can keep both the original dataset and also the modified dataset in the working environment.

In the next step the reader should attempt to separate survivor and nonsurvivor data from the modified dataset.

```
titanic_survivor = titanic_train_droppedna[titanic_train_droppedna$Survived ==1, ]
```

```
titanic_nonsurvivor = titanic_train_droppedna[titanic_train_droppedna$Survived == 0,]
```

This is the time for the user to generate some graphs from the data.

This is the time for the user to generate some graphs from the data.

Creating bar chart:

```
barplot(table(titanic_survivor$Sex))
```

```
barplot(table(titanic_nonsurvivor$Sex))
```

Creating a Histogram:

```
hist(titanic_survivor$Age)
```

```
hist(titanic_nonsurvivor$Age)
```

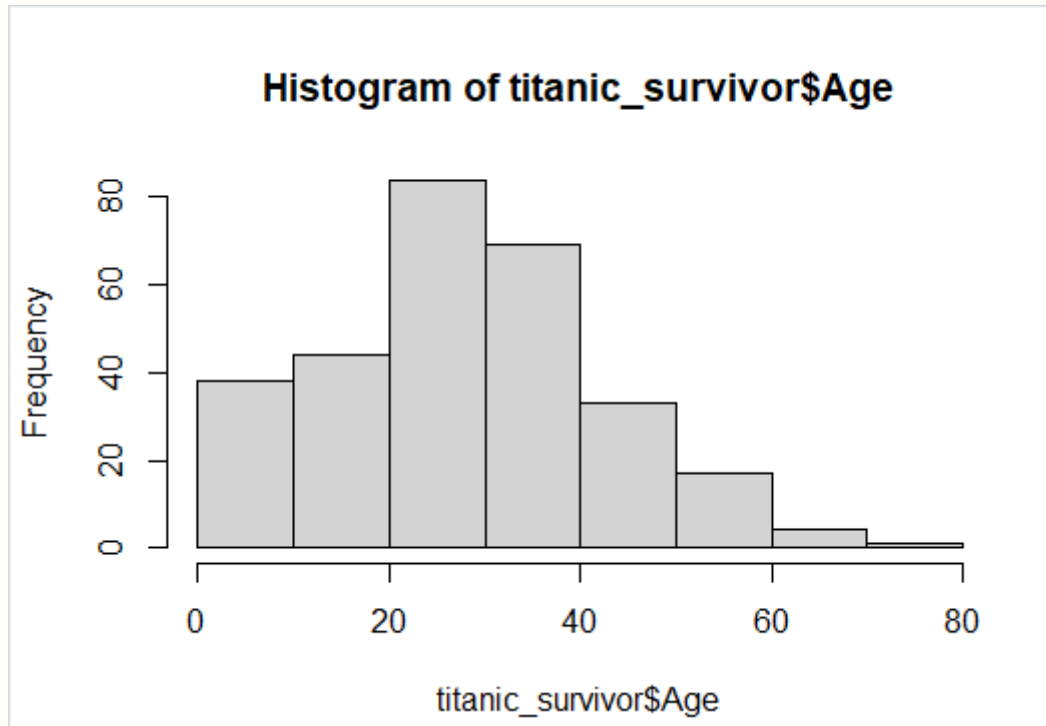


Image showing Histogram created from survivor details displayed

Exploratory data analysis:

This is a statistical technique used to analyze data sets in order to summarize their important main characteristics generally using visual aids. The following aspects of the data set can be studied using this approach:

1. Main characteristics or features of the data.
2. The variables and their relationships.
3. Finding out the important variables that can be used in the problem.

This is an interactive approach that includes:

Generating questions about the data.

Searching for answers using visualization, transformation, and modeling of the data.

Using the lessons that has been learnt in order to refine the set of questions or to generate a new set of questions.

Before actually using exploratory data analysis, one must perform a proper data inspection. In this example the author will be using loafercreek dataset from the soilDB package that can be installed as a library in R. Most wonderful thing about R is that one can install various datasets in the form of libraries in order to create a learning environment that can be used for training purposes.

Before proceeding any further the user needs to install the following packages:

aqp package

ggplot2 package

soilDB package

These packages can be installed from the R console using the `install.packages()` command and can be loaded into the script by using the `library()` command.

```
install.packages("aqp")
```

```
install.packages("ggplot2")
```

```
install.packages("soilDB")
```

```
library("aqp")
```

```
library("ggplot2")
```

```
library("soilDB")
```

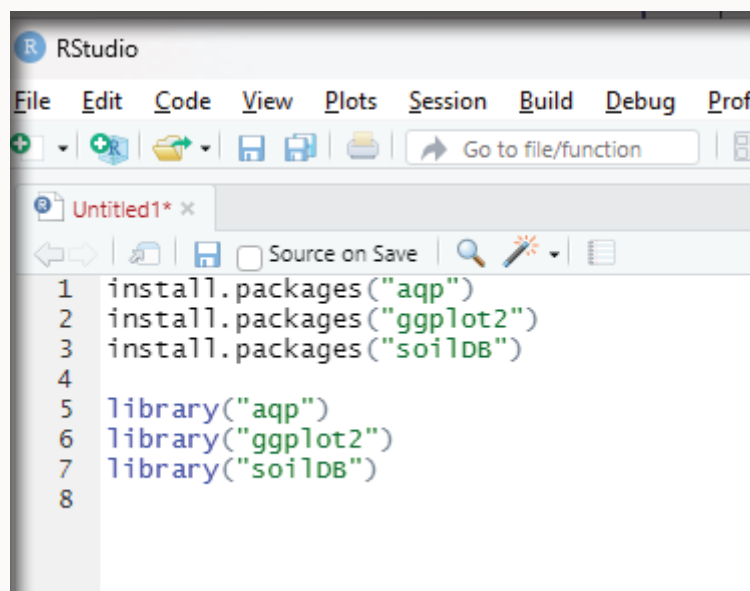


Image showing the codes for installation and loading the packages entered into scripting window

```

# Data Inspection in EDA
# loading the required packages
library(aqp)
library(soilDB)

# Load from the loafercreek dataset
data("loafercreek")

# Construct generalized horizon designations
n <- c("A", "BA", "Bt1", "Bt2", "Cr", "R")

# REGEX rules
p <- c("A", "BA|AB", "Bt|Bw", "Bt3|Bt4|2B|C",
      "Cr", "R")

# Compute genhz labels and
# add to loafercreek dataset
loafercreek$genhz <- generalize.hz(
  loafercreek$hzname,
  n, p)

# Extract the horizon table
h <- horizons(loafercreek)

# Examine the matching of pairing of
# the genhz label to the hznames
table(h$genhz, h$hzname)

vars <- c("genhz", "clay", "total_frgs_pct",
          "phfield", "effclass")
summary(h[, vars])

sort(unique(h$hzname))
h$hzname <- ifelse(h$hzname == "BT",
                  "Bt", h$hzname)

```

Output:

```
> table(h$genhz, h$hzname)
```

	2BCt	2Bt1	2Bt2	2Bt3	2Bt4	2Bt5	2CB	2CBt	2Cr	2Crt	2R	A	A1	A2	AB	ABt	Ad	Ap	B	BA	BAt	BC	BCt	Bt
	Bt1	Bt2	Bt3	Bt4	Bw	Bw1	Bw2	Bw3	C															
A		0	0	0	0	0	0	0	0	0	0	97	7	7	0	0	1	1	0	0	0	0	0	0
BAt		0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	31	8	0	0	0	0
Bt1		0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	0	0	0	0	0	8	94	89
Bt2		1	2	7	8	6	1	1	1	0	0	0	0	0	0	0	0	0	0	5	16	0	0	0
Cr		0	0	0	0	0	0	0	0	4	2	0	0	0	0	0	0	0	0	0	0	0	0	0
R		0	0	0	0	0	0	0	0	0	5	0	0	0	0	0	0	0	0	0	0	0	0	0
not-used		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0

	CBt	Cd	Cr	Cr/R	Crt	H1	Oi	R	Rt
A	0	0	0	0	0	0	0	0	0
BAt	0	0	0	0	0	0	0	0	0
Bt1	0	0	0	0	0	0	0	0	0
Bt2	6	1	0	0	0	0	0	0	0
Cr	0	0	49	0	20	0	0	0	0
R	0	0	0	1	0	0	0	41	1
not-used	0	0	0	0	0	1	24	0	0

```
> summary(h[, vars])
```

	genhz	clay	total_frgs_pct	phfield	effclass
A	:113	Min. :10.00	Min. :0.00	Min. :4.90	very slight: 0
BAt	:40	1st Qu.:18.00	1st Qu.:0.00	1st Qu.:6.00	slight : 0
Bt1	:208	Median :22.00	Median :5.00	Median :6.30	strong : 0
Bt2	:116	Mean :23.67	Mean :14.18	Mean :6.18	violent : 0
Cr	:75	3rd Qu.:28.00	3rd Qu.:20.00	3rd Qu.:6.50	none :86
R	:48	Max. :60.00	Max. :95.00	Max. :7.00	NA's :540
not-used:	26	NA's :173		NA's :381	

```
> sort(unique(h$hzname))
```

```
[1] "2BCt" "2Bt1" "2Bt2" "2Bt3" "2Bt4" "2Bt5" "2CB" "2CBt" "2Cr" "2Crt" "2R" "A" "A1" "A2" "AB"
"ABt" "Ad" "Ap" "B"
[20] "BA" "BAt" "BC" "BCt" "Bt" "Bt1" "Bt2" "Bt3" "Bt4" "Bw" "Bw1" "Bw2" "Bw3" "C" "CBt"
"Cd" "Cr" "Cr/R" "Crt"
[39] "H1" "Oi" "R" "Rt"
```

Descriptive statistics:

Measures of central tendency

Measures of dispersion

Correlation

Measures of central tendency:

This is a feature of descriptive statistics. This tells about how the group of data is clustered around the central value of the distribution. Central tendency performs the following measures:

Arithmetic mean

Geometric mean

Harmonic mean

Median

Mode

Arithmetic mean can be calculated by mean() function.

syntax: mean(x, trim, na.rm=FALSE)

x = object

trim = specifies number of values to be removed from each side of the object before calculating the mean.

The value is between 0 to 0.5.

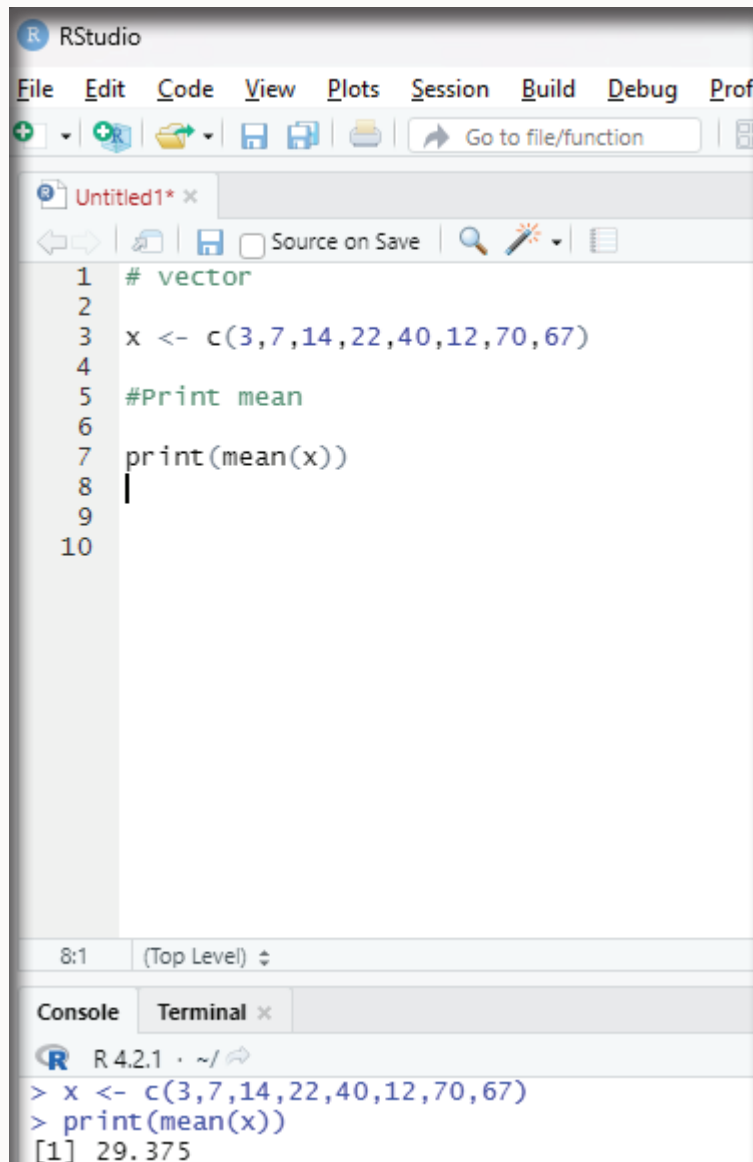
na.rm = If true it removes NA value from x.

vector

x <- c(3,7,14,22,40,12,70,67)

#Print mean

print(mean(x))



The image shows the RStudio interface. The source editor contains the following R code:

```
1 # vector
2
3 x <- c(3,7,14,22,40,12,70,67)
4
5 #Print mean
6
7 print(mean(x))
8
9
10
```

The console at the bottom shows the execution of the code:

```
R 4.2.1 ~/  
> x <- c(3,7,14,22,40,12,70,67)  
> print(mean(x))  
[1] 29.375
```

Image showing Arithmetic mean calculated for a vector

Using trim and na.rm function:

Trimmed mean is a dataset's mean that is determined after deleting a certain percentage of the dataset's smallest and greatest values. N.A value is also ignored.

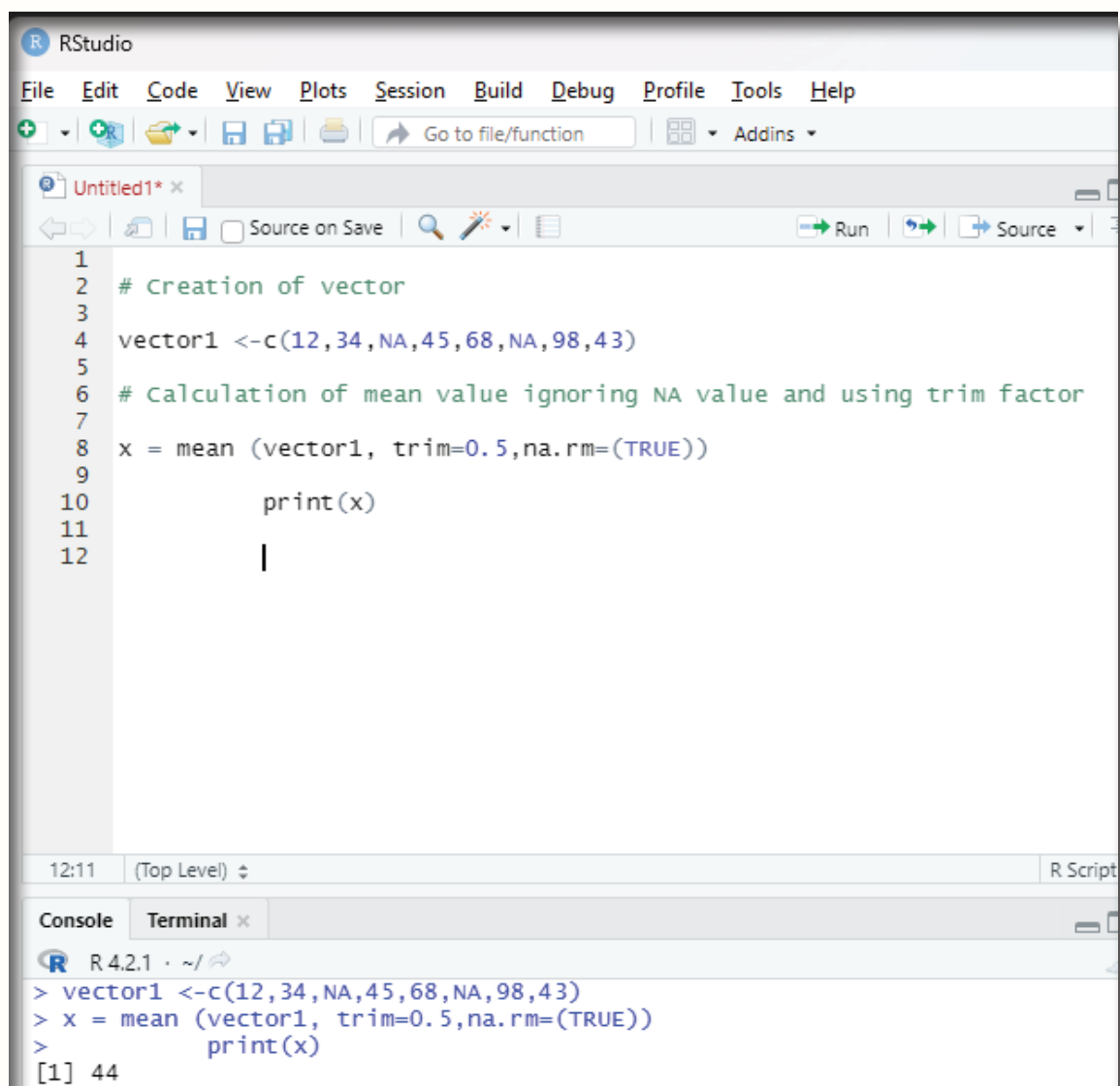
```
# Creation of vector
```

```
vector1 <-c(12,34,NA,45,68,NA,98,43)
```

```
# Calculation of mean value ignoring NA value and using trim factor
```

```
x = mean (vector1, trim=0.5,na.rm=(TRUE))
```

```
print(x)
```



The screenshot shows the RStudio interface. The top menu bar includes File, Edit, Code, View, Plots, Session, Build, Debug, Profile, Tools, and Help. Below the menu is a toolbar with icons for file operations and a search bar. The main editor window, titled 'Untitled1*', contains the following R code:

```
1  
2 # Creation of vector  
3  
4 vector1 <-c(12,34,NA,45,68,NA,98,43)  
5  
6 # Calculation of mean value ignoring NA value and using trim factor  
7  
8 x = mean (vector1, trim=0.5,na.rm=(TRUE))  
9  
10     print(x)  
11  
12     |
```

The bottom panel shows the Console window with the following output:

```
R 4.2.1 ~/  
> vector1 <-c(12,34,NA,45,68,NA,98,43)  
> x = mean (vector1, trim=0.5,na.rm=(TRUE))  
>     print(x)  
[1] 44
```

Image showing the use of trim and na.rm functions

Geometric mean:

This type of mean is computed by multiplying all the data values and thus, shows the central tendency for given data distribution.

prod() and length() functions help in finding the geometric mean of a given set of numbers in a vector.

Syntax:

$\text{prod}(x)^{(1/\text{length}(x))}$

prod() function returns the product of all values present in vector x.

length() function returns the length of the vector x.

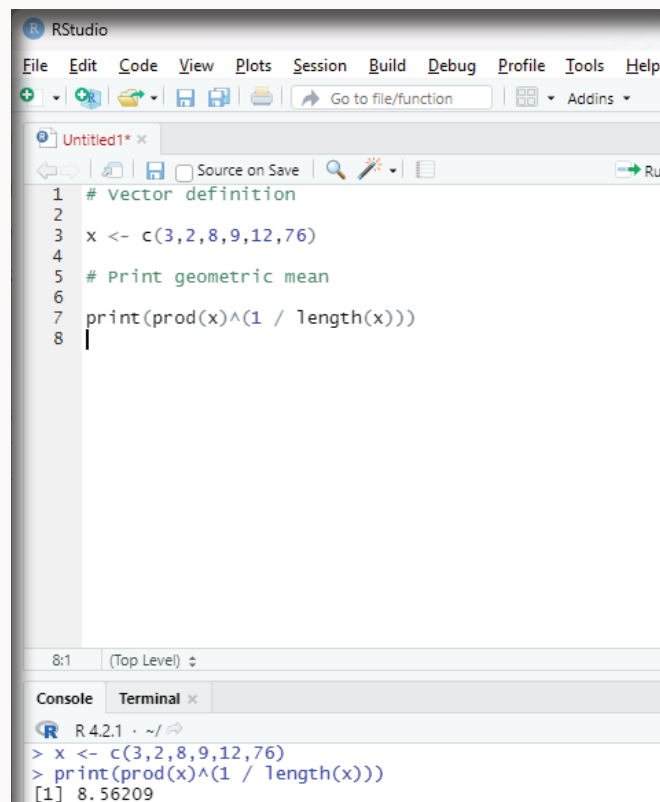
Code:

```
# Vector definition
```

```
x <- c(3,2,8,9,12,76)
```

```
# Print geometric mean
```

```
print(prod(x)^(1 / length(x)))
```

A screenshot of the RStudio interface. The top menu bar includes File, Edit, Code, View, Plots, Session, Build, Debug, Profile, Tools, and Help. Below the menu is a toolbar with icons for saving, running, and other functions. The main editor window, titled 'Untitled1*', contains the following R code:

```
1 # vector definition
2
3 x <- c(3,2,8,9,12,76)
4
5 # Print geometric mean
6
7 print(prod(x)^(1 / length(x)))
8
```

The bottom panel of RStudio is split into 'Console' and 'Terminal' tabs. The 'Console' tab is active and shows the output of the R code:

```
R 4.2.1 ~ /
> x <- c(3,2,8,9,12,76)
> print(prod(x)^(1 / length(x)))
[1] 8.56209
```

Image showing Arithmetic mean calculated

This is another type of mean that is used as a measure of central tendency. It is computed as reciprocal of the arithmetic mean of reciprocals of the given set of values.

```
# Defining the vector
```

$$x <- c(3,6,8,9)$$

```
# Print harmonic mean
```

```
print(1 / mean(1 / x))
```

Image showing calculation of Harmonic mean

Median:

Median value in statistics is a measure of central tendency which represents the middle most value of a given set of values.

Syntax:

`median(x, na.rm=FALSE)`

Parameters:

x: It is the data vector

na.rm: If TRUE then it removes the NA values from x.

Defining a vector:

x <- c(3,7,8,90,85,43)

Print median

median(x)

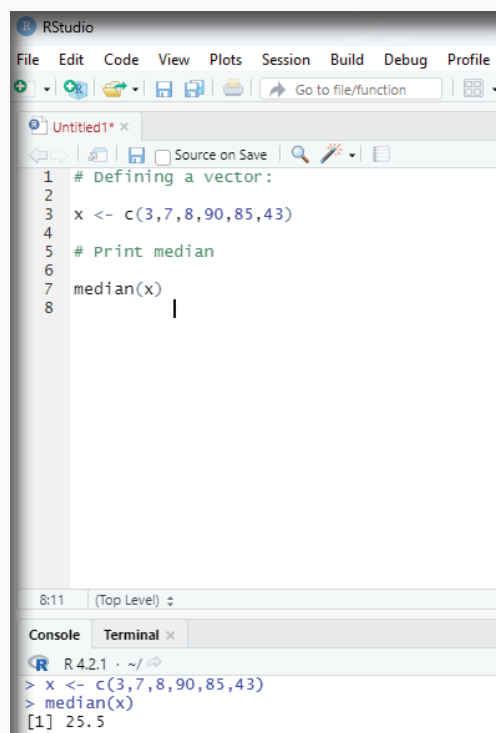


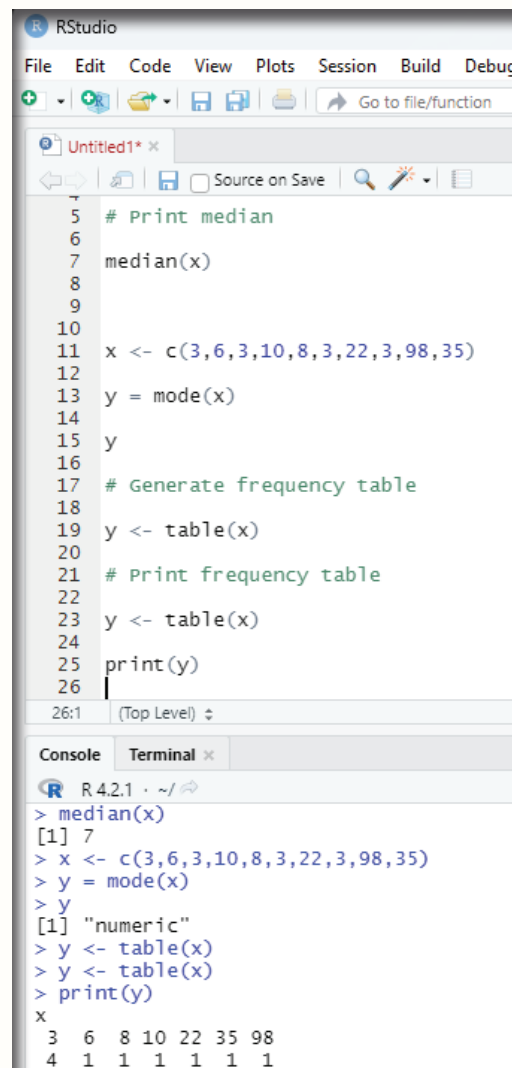
Image showing median value calculation

Mode:

Mode of a given set of values is the value that is repeated the most in a dataset. There could be multiple mode values if there are two or more values with matching maximum frequency

Single mode value:

```
# Defining vector
x <- c(3,6,3,10,8,3,22,3,98,35)
y = mode(x)
# Generate frequency table
y <- table(x)
# Print frequency table
y = <- table(x)
print(y)
```



The screenshot shows the RStudio interface with a script editor and a console. The script editor contains the following code:

```
5 # Print median
6
7 median(x)
8
9
10
11 x <- c(3,6,3,10,8,3,22,3,98,35)
12
13 y = mode(x)
14
15 y
16
17 # Generate frequency table
18
19 y <- table(x)
20
21 # Print frequency table
22
23 y <- table(x)
24
25 print(y)
26
```

The console shows the output of the code:

```
> median(x)
[1] 7
> x <- c(3,6,3,10,8,3,22,3,98,35)
> y = mode(x)
> y
[1] "numeric"
> y <- table(x)
> y <- table(x)
> print(y)
x
 3  6  8 10 22 35 98
4  1  1  1  1  1  1
```

Image showing frequency of the data within a vector

```
# Multiple mode values
```

```
# Defining vector
```

```
x <- c(3,6,7,3,4,23,6,4,23,76,87,76,4,3,4)
```

```
# Generate frequency table
```

```
y <- table(x)
```

```
# Print frequency table
```

```
print(y)
```

```
# Mode of x
```

```
m <- names(y)[which(y == max(y))]
```

```
# Print mode
```

```
print(m)
```

Skewness and Kurtosis in R Programming:

In statistical analysis, skewness and kurtosis are the measures that reveals the shape of the data distribution. Both of these parameters are numerical methods to analyze the shape of the data set.

Skewness - This is a statistical numerical method to measure the asymmetry of the distribution of the data set. It reveals the position of the majority of data values in the distribution around the mean value.

Positive skew - If the coefficient of skewness is greater than 0, then the graph is said to be positively skewed with the majority of data values less than the mean. Most of the values are concentrated on the left side of the graph.

Package moments needs to be installed.

```
install.packages("moments")
```

```
# Required for skewness() function
```

```
library(moments)
```

```
# Defining the data vector
```

The image shows the RStudio interface. The script editor contains the following R code:

```
1 # Multiple mode values
2
3 # Defining vector
4
5 x <- c(3,6,7,3,4,23,6,4,23,76,87,76,4,3,4)
6
7 # Generate frequency table
8
9 y <- table(x)
10
11 # Print frequency table
12
13 print(y)
14
15 # Mode of x
16
17 m <- names(y)[which(y == max(y))]
18
19 # Print mode
20
21
22 print(m)
```

The console output shows the execution of the code:

```
> x <- c(3,6,7,3,4,23,6,4,23,76,87,76,4,3,4)
> y <- table(x)
> print(y)
x
 3  4  6  7 23 76 87
 3  4  2  1  2  2  1
> m <- names(y)[which(y == max(y))]
> print(m)
[1] "4"
```

Image showing multiple modes calculation

```
x <- c(40, 41, 42, 43, 50)
```

```
# Print skewness of distribution
```

```
print(skewness(x))
```

```
# Histogram of distribution
```

```
hist(x)
```

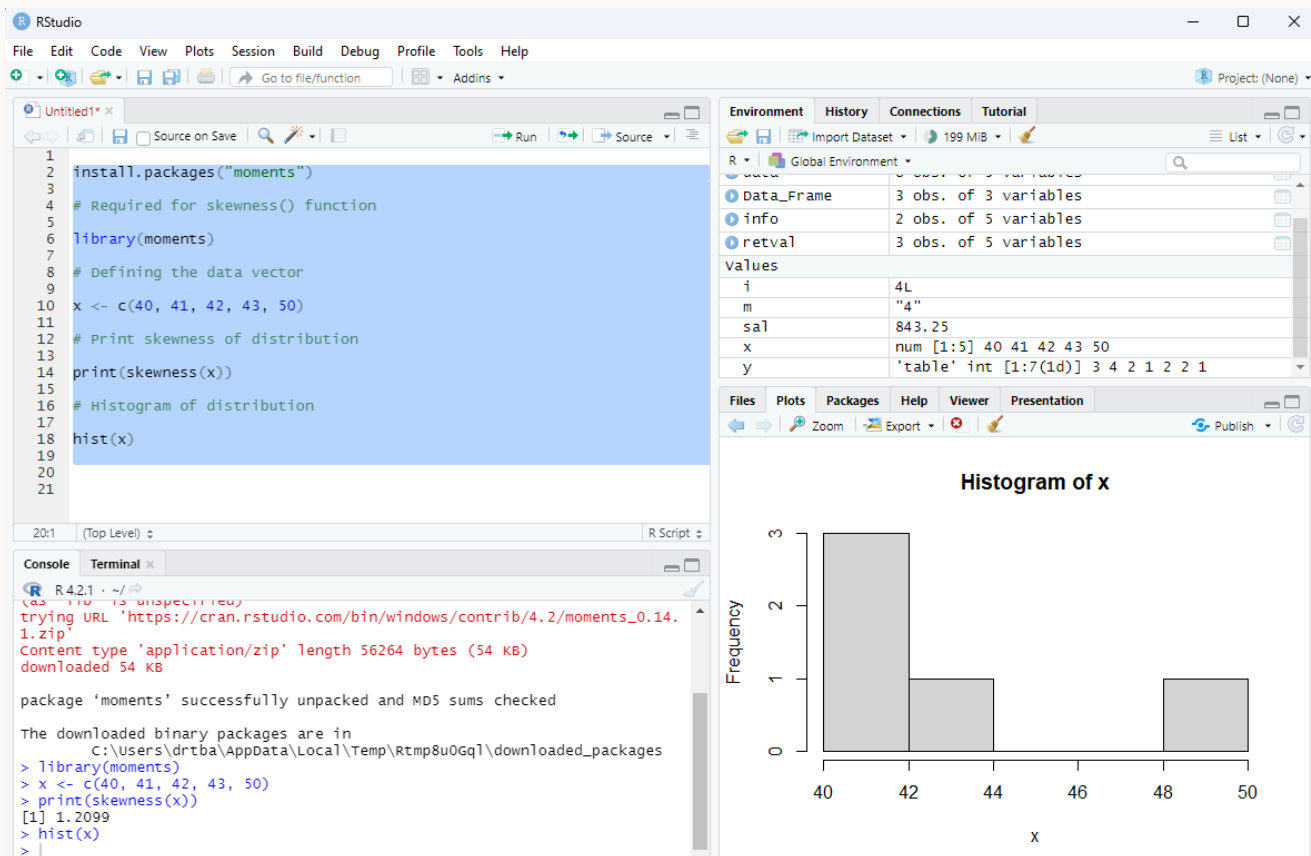


Image showing skewness calculated

Zero skewness or symmetric:

If the coefficient of skewness is equal to 0 or close to 0 then the graph is symmetric and data is normally distributed.

Defining normally distributed data vector

x <- rnorm(50, 10, 10)

Print skewness of distribution

print(skewness(x))

Histogram of distribution

hist(x)

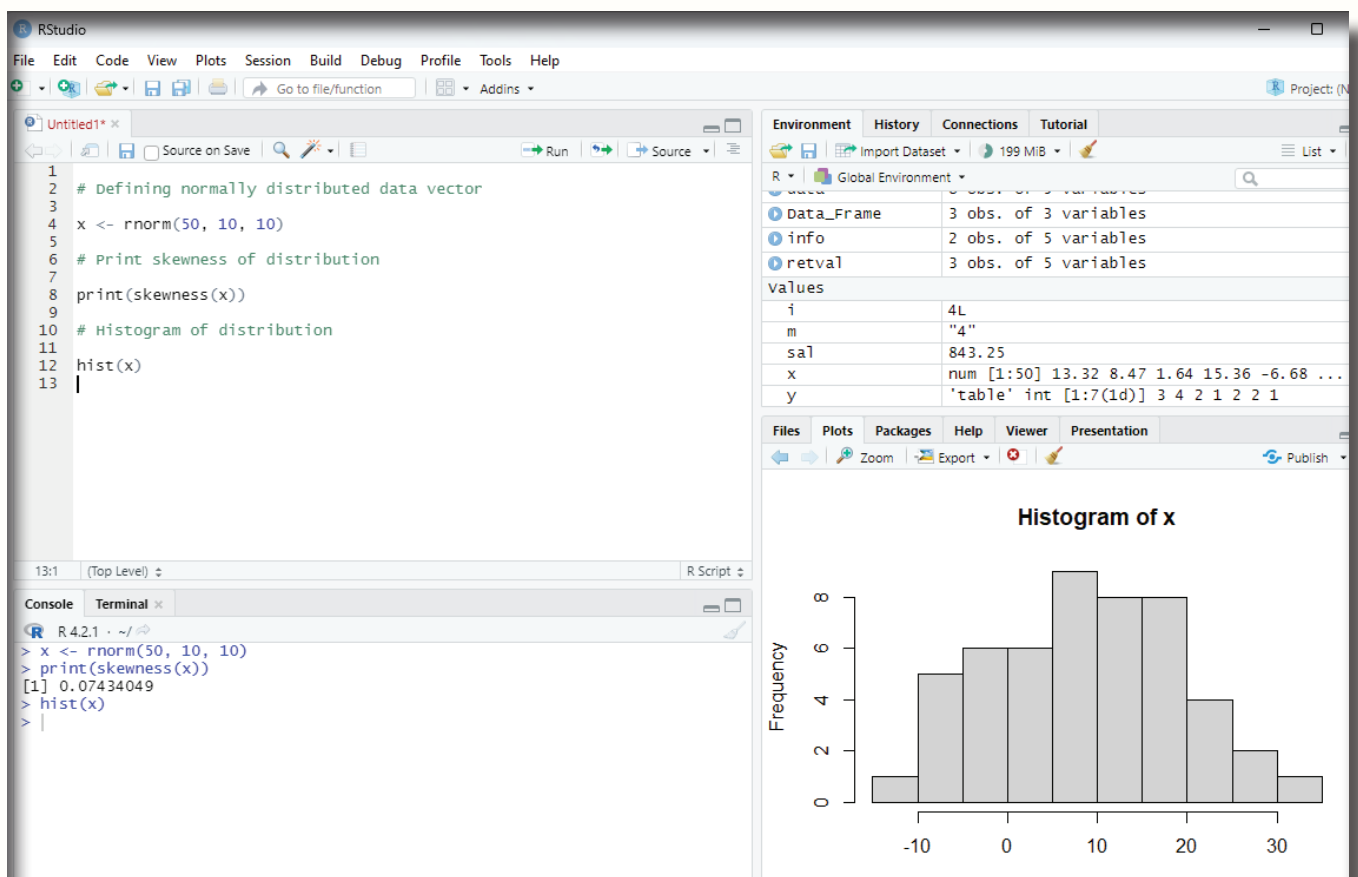


Image showing zero skewness

Negatively skewed:

If the coefficient of skewness is less than 0 then it is negatively skewed with the majority of data values greater than mean.

```
# Defining data vector
```

```
x <- c(10,11,21,22,23,25)
```

```
# Print skewness of distribution
```

```
print(skewness(x))
```

```
# Histogram of distribution
```

```
hist(x)
```

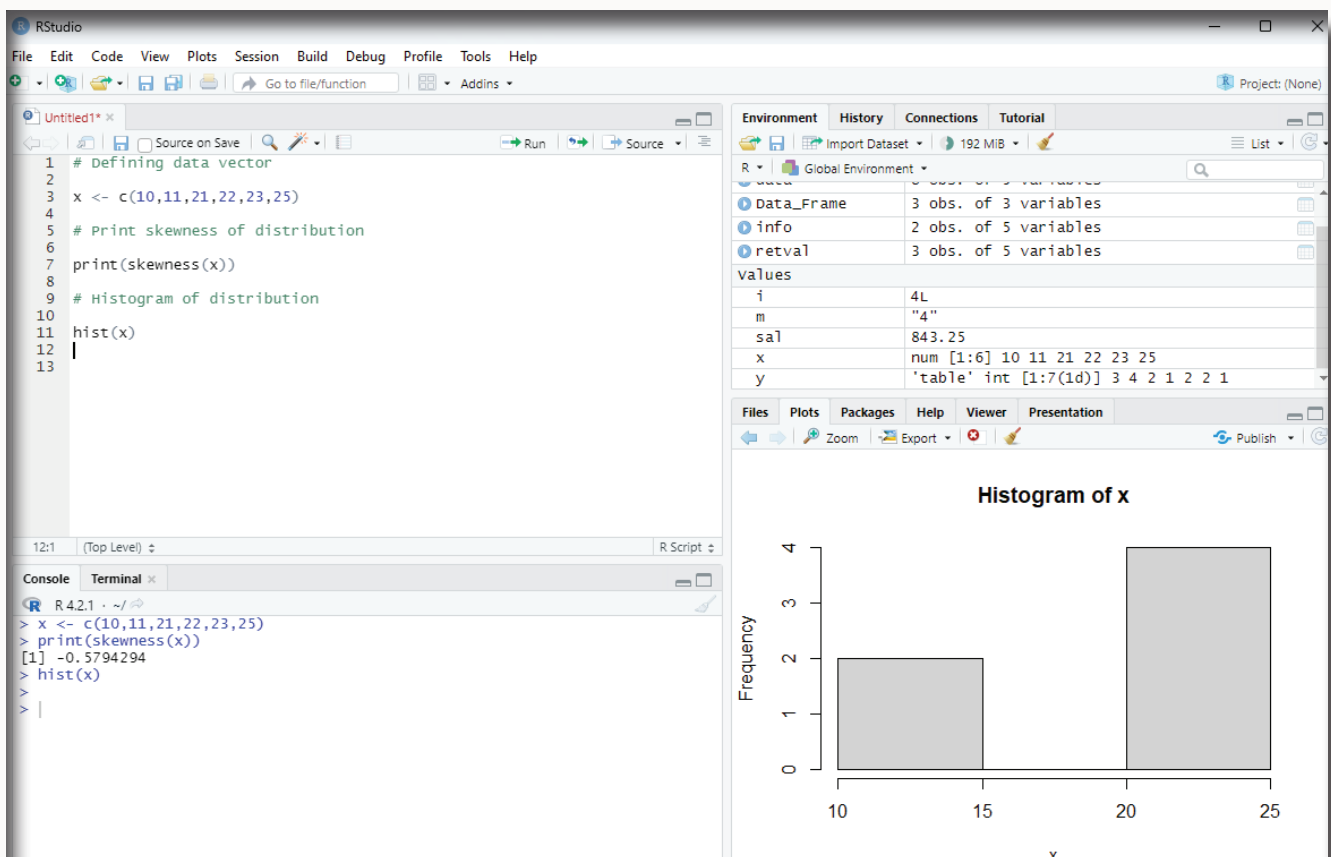


Image showing negatively skewed data

Kurtosis:

This is a numerical method in statistics that measure the sharpness of the peak in the data distribution.

There are three types of kurtosis:

Platykurtic - If the coefficient of kurtosis is less than 3 then the data distribution is platykurtic. Being platykurtic doesn't mean that the graph is flat topped.

Mesokurtic - If the coefficient of kurtosis is equal to 3 or close to 3 then the data distribution is mesokurtic. For normal distribution kurtosis value is approximately equal to 3.

Leptokurtic - If the coefficient is greater than 3 then the data distribution is leptokurtic and shows a sharp peak on the graph.

Example for platykurtic distribution

```
# Defining data vector
```

```
x <- c(rep(61, each = 10), rep(64, each = 18),  
rep(65, each = 23), rep(67, each = 32), rep(70, each = 27),  
rep(73, each = 17))
```

```
# Print skewness of distribution  
print(kurtosis(x))
```

```
# Histogram of distribution  
hist(x)
```

Example for mesokurtic data set:

```
# Defining data vector  
x <- rnorm(100)
```

```
# Print skewness of distribution  
print(kurtosis(x))
```

```
# Histogram of distribution  
hist(x)
```

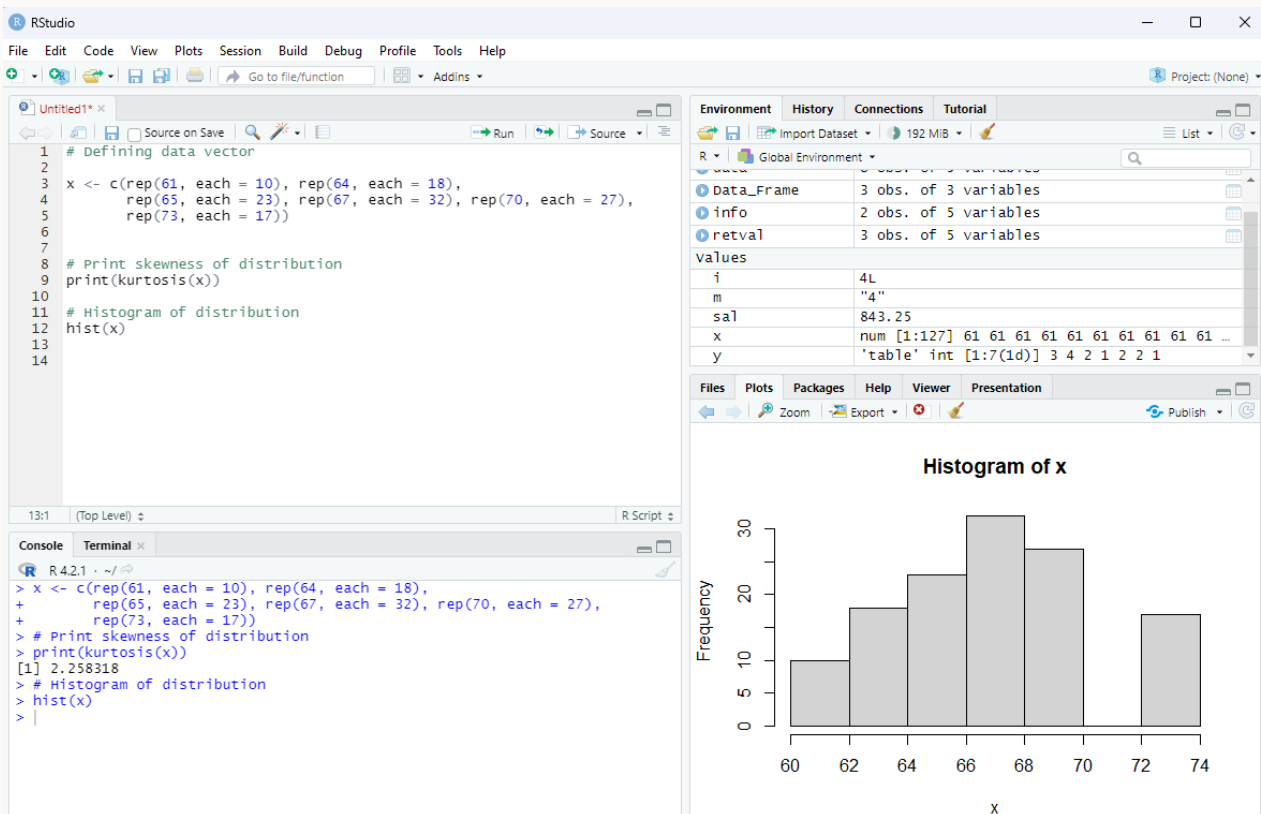


Image showing platykurtic distribution

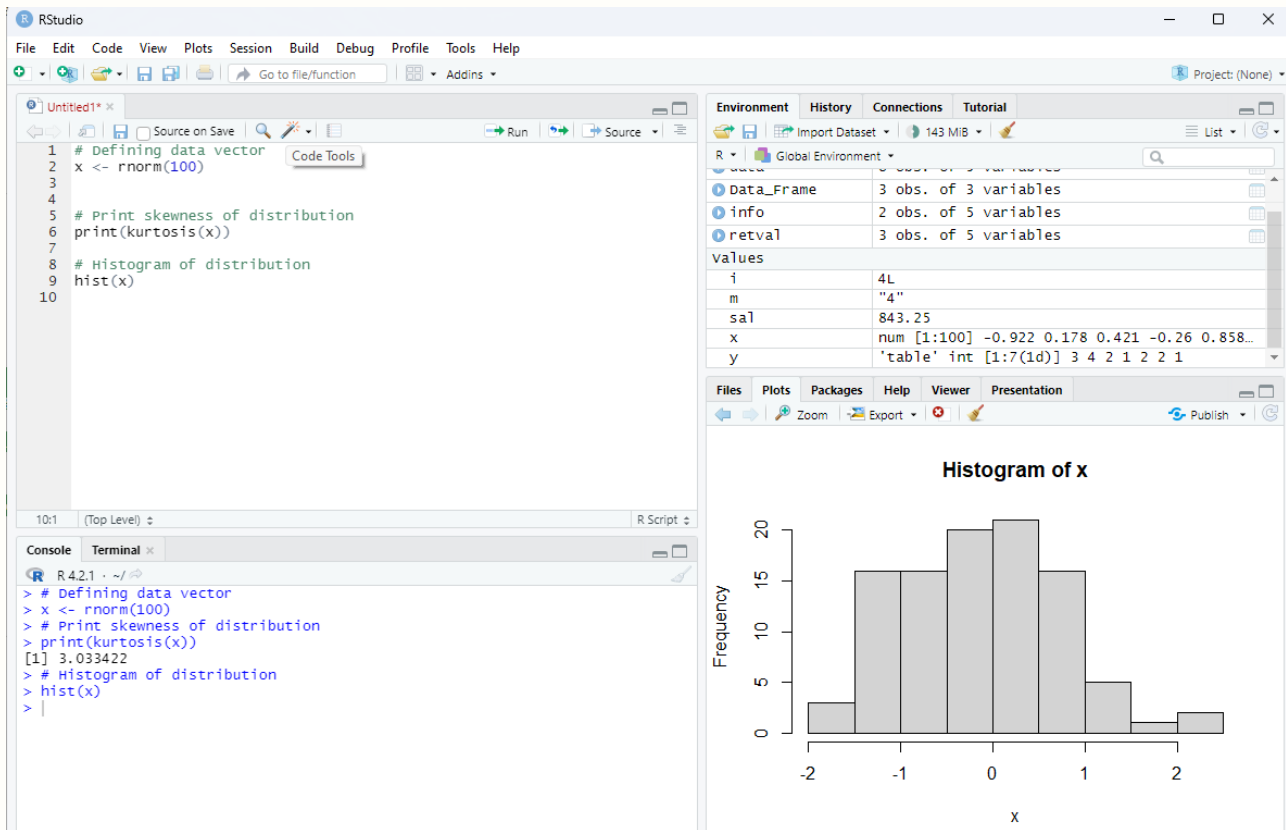


Image showing display of mesokurtic data

Leptokurtic distribution Example:

Defining data vector

*x <- c(rep(61, each = 2), rep(64, each = 5),
rep(65, each = 42), rep(67, each = 12), rep(70, each = 10))*

Print skewness of distribution

print(kurtosis(x))

Histogram of distribution

hist(x)

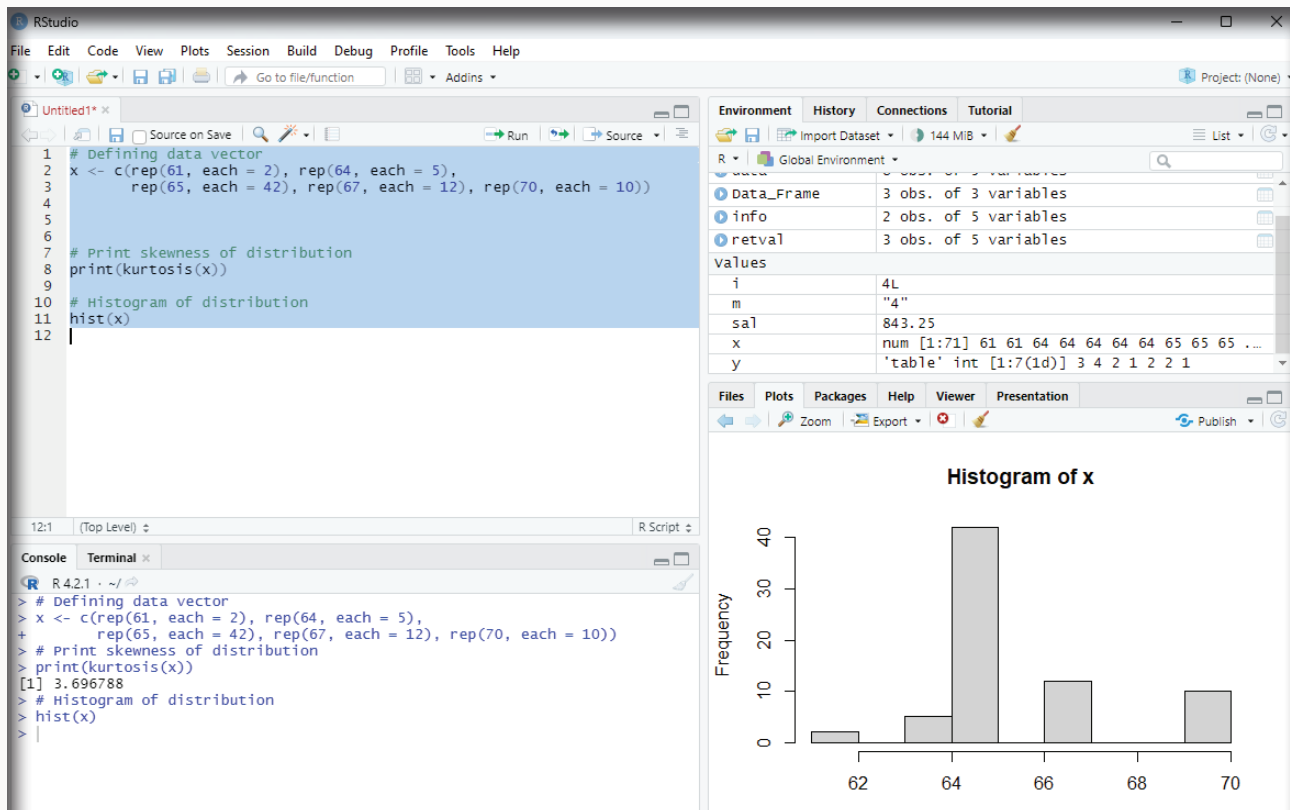


Image showing Leptokurtic distribution

Hypothesis Testing in R Programming

Hypothesis is made by the researchers about the data collected. Hypothesis is an assumption made by the researchers and it need not be true. R Programming can be used to test and validate the hypothesis of a researcher. Based on the results of calculation the hypothesis can be branded as true or can be rejected. This concept is known as Statistical Inference.

Hypothesis testing is a 4 step process:

State the hypothesis - This step is begun by stating the null and alternate hypothesis which is presumed to be true.

Formulate an analysis plan and set the criteria for decision - In this step the significance level of the test is set. The significance level is the probability of a false rejection of a hypothesis.

Analyze sample data - In this, a test statistic is used to formulate the statistical comparison between the sample mean and the mean of the population or standard deviation of the sample and standard deviation of the population.

Interpret decision - The value of test statistic is used to make the decision based on the significance level. For example, if the significance level is set to 0.1 probability, then the sample mean less than 10% will be rejected. Otherwise the hypothesis is retained as true.

One Sample T-Testing:

This approach collects a huge amount of data and tests it on random samples. In order to perform T-Test in R, normally distributed data is required. This test is used to ascertain the mean of the sample with the population. For example, the weight of persons living in an area is different or identical to other persons living in other areas.

Syntax:

```
t.test(x, mu)
```

x - represents the numeric vector of data.

mu - represents true value of the mean.

One can ascertain more optional parameters of t.test by the following command:

```
help("t.test")
```

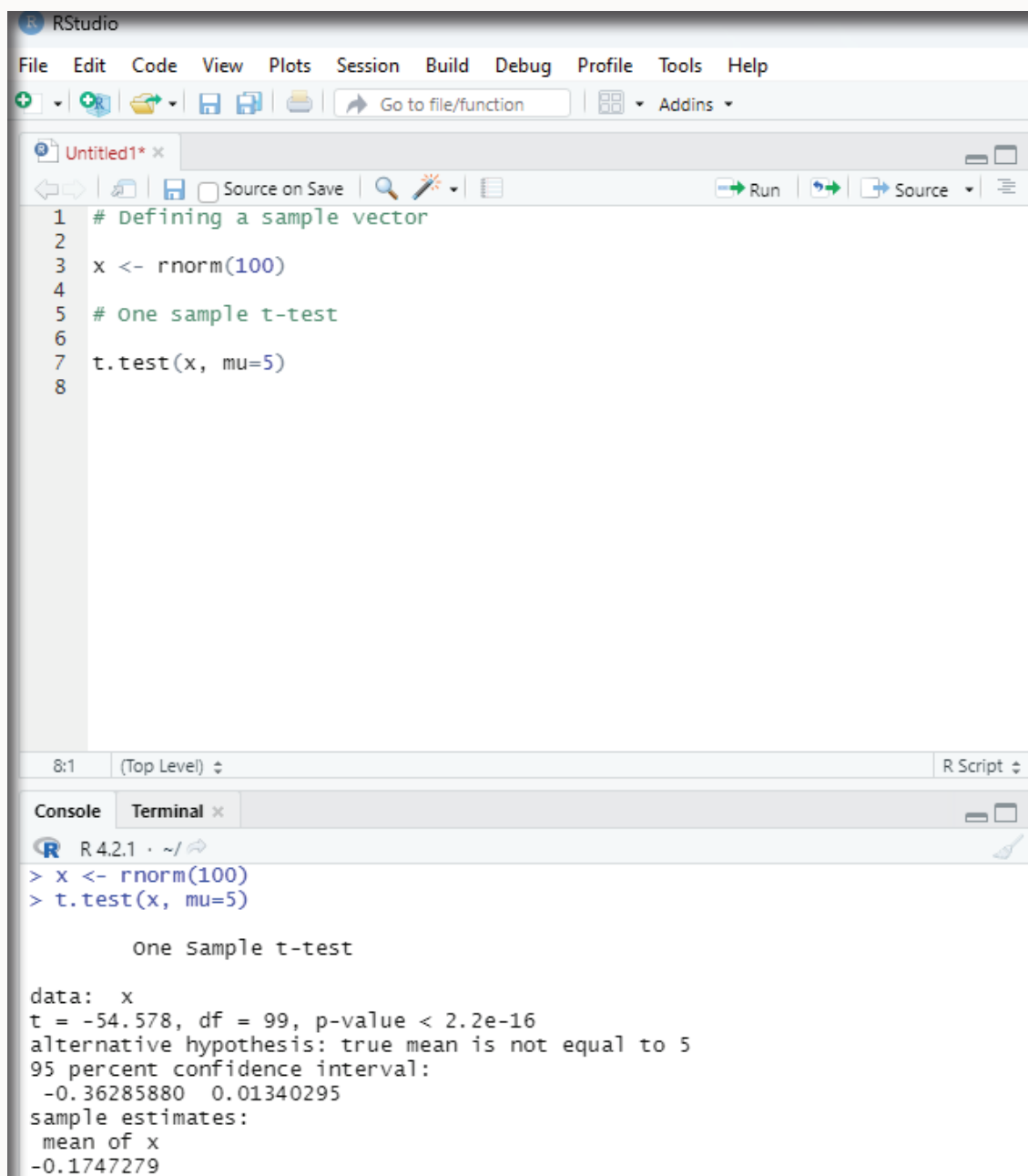
Example:

Defining a sample vector

x <- rnorm(100)

One sample t-test

t.test(x, mu=5)



The screenshot shows the RStudio interface. The script editor contains the following code:

```
1 # Defining a sample vector
2
3 x <- rnorm(100)
4
5 # One sample t-test
6
7 t.test(x, mu=5)
8
```

The console output shows the results of the t-test:

```
R 4.2.1 ~ /
> x <- rnorm(100)
> t.test(x, mu=5)

    One Sample t-test

data:  x
t = -54.578, df = 99, p-value < 2.2e-16
alternative hypothesis: true mean is not equal to 5
95 percent confidence interval:
 -0.36285880  0.01340295
sample estimates:
mean of x
-0.1747279
```

Image showing one sample t-testing

The R function `rnorm` generates a vector of normally distributed random numbers. `rnorm` can take up to 3 arguments:

`n` - the number of random variables to generate

`mean` - if not specified it takes a default value of 0.

`sd` - Standard deviation. If not specified it takes a default value of 1.

Example:

```
n <- rnorm(100000, mean = 100, sd = 36)
```

Two Sample T-Testing:

In two sample T-Testing, the sample vectors are compared. If `var.equal = TRUE`, the test assumes that the variances of both the samples are equal.

Syntax:

```
t.test(x,y)
```

Parameters:

`x` and `y` : numeric vectors

```
# Defining sample vector
```

```
x <- rnorm(100)
```

```
y <- rnorm(100)
```

```
# Two Sample T-Test
```

```
t.test(x, y)
```

The image shows the RStudio interface. The top menu bar includes File, Edit, Code, View, Plots, Session, Build, Debug, Profile, Tools, and Help. Below the menu is a toolbar with icons for file operations and a search bar. The main editor window, titled 'Untitled1*', contains the following R code:

```
1 # Defining sample vector
2
3 x <- rnorm(100)
4 y <- rnorm(100)
5
6 # Two Sample T-Test
7
8 t.test(x, y)
9
```

Below the editor is a status bar showing '9:1 (Top Level)' and 'R Script'. The bottom pane has two tabs: 'Console' and 'Terminal'. The 'Console' tab is active and displays the output of the executed code:

```
R 4.2.1 ~ /
> x <- rnorm(100)
> y <- rnorm(100)
> t.test(x, y)

      welch Two Sample t-test

data:  x and y
t = -1.2136, df = 194.15, p-value = 0.2264
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -0.4865683  0.1158761
sample estimates:
 mean of x   mean of y 
-0.15818864  0.02715747
```

Image showing Two sample T-Testing

Directional Hypothesis:

This is used when the direction of the hypothesis can be specified. This is ideal if the user desires to know the sample mean is lower or greater than another mean of sample data.

Syntax:

```
t.test(x,mu,alternative)
```

Parameters:

x - represents numeric vector data

mu - represents mean against which sample data has to be tested

alternative - Sets the alternative hypothesis.

Example:

```
# Defining sample vector
```

```
x <- rnorm(100)
```

```
# Directional hypothesis testing
```

```
t.test(x, mu=2, alternative = 'greater')
```

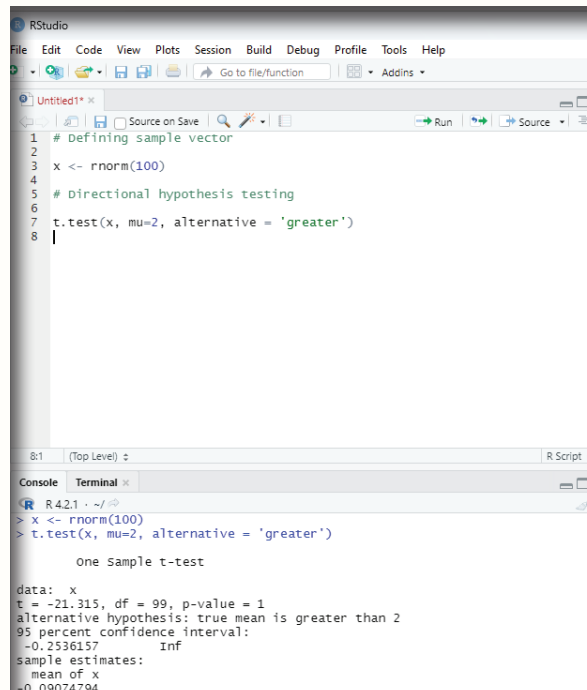


Image showing directional hypothesis testing

One Sample Mu test:

This test is used when comparison has to be computed on one sample and the data is non-parametric. It is performed using `wilcox.test()` function in R programming.

Syntax:

```
wilcox.test(x,y,exact=NULL)
```

`x` and `y` : represents numeric vector

`exact`: represents logical value which indicates whether p-value be computed.

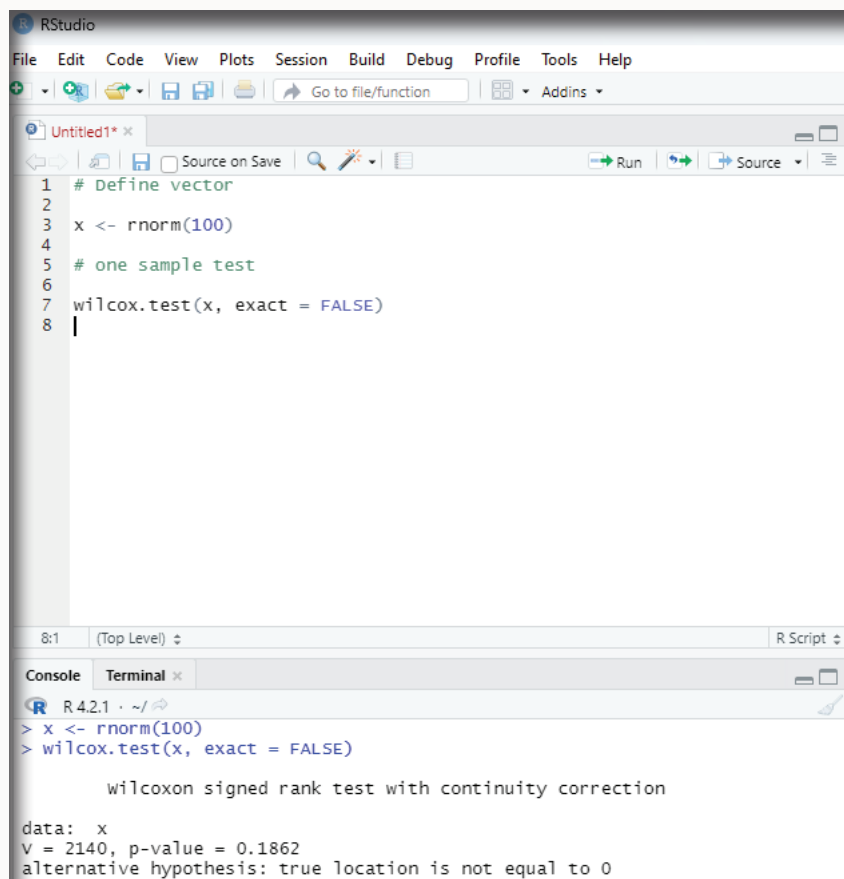
Example:

```
# Define vector
```

```
x <- rnorm(100)
```

```
# one sample test
```

```
wilcox.test(x, exact = FALSE)
```

A screenshot of the RStudio interface. The top menu bar includes File, Edit, Code, View, Plots, Session, Build, Debug, Profile, Tools, and Help. Below the menu is a toolbar with icons for file operations and running code. The main editor window, titled 'Untitled1*', contains the following R code:

```
1 # Define vector
2
3 x <- rnorm(100)
4
5 # one sample test
6
7 wilcox.test(x, exact = FALSE)
8 |
```

The bottom panel shows the 'Console' tab with the output of the code execution:

```
R 4.2.1 ~ /
> x <- rnorm(100)
> wilcox.test(x, exact = FALSE)

      wilcoxon signed rank test with continuity correction

data:  x
V = 2140, p-value = 0.1862
alternative hypothesis: true location is not equal to 0
```

Image showing wilcox test

Two sample Mu-Test:

This test is performed to compare two samples of data.

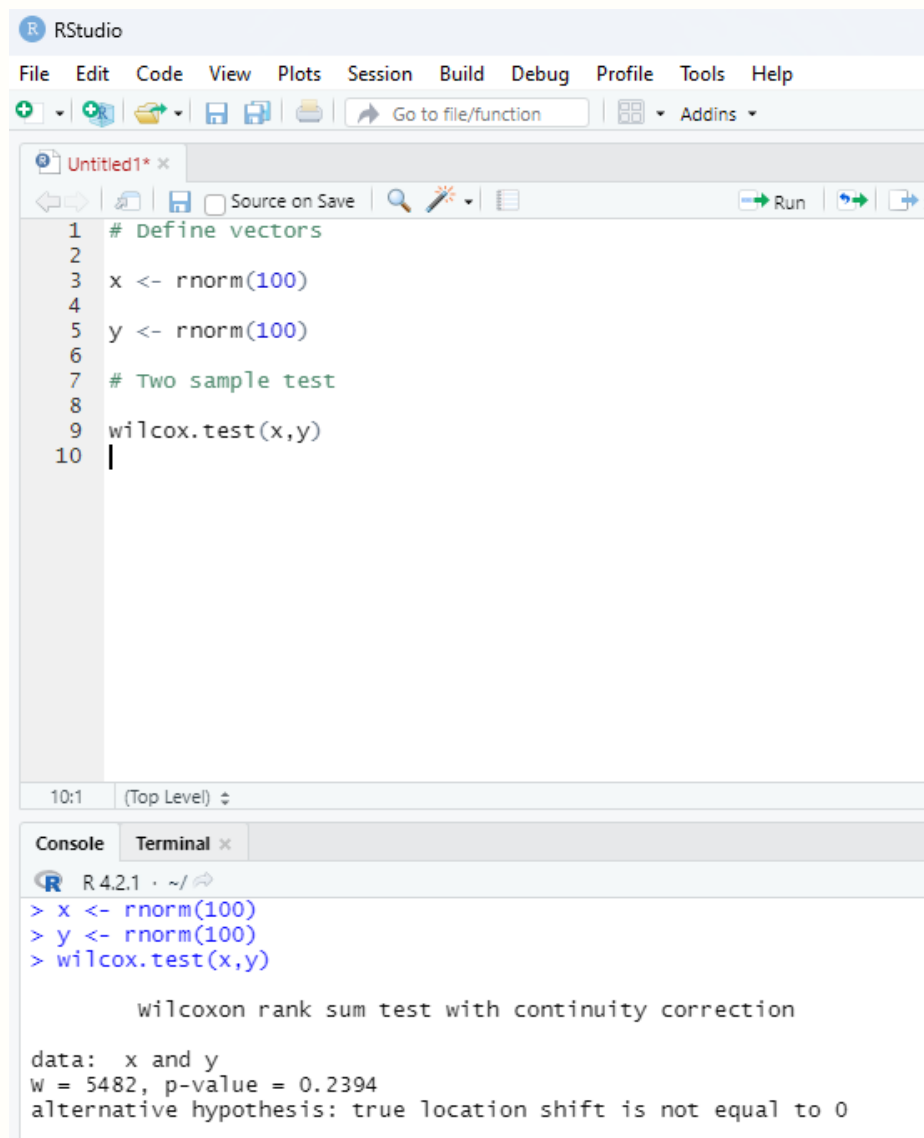
```
# Define vectors
```

```
x <- rnorm(100)
```

```
y <- rnorm(100)
```

```
# Two sample test
```

```
wilcox.test(x,y)
```



The screenshot displays the RStudio interface. The script editor shows the following R code:

```
1 # Define vectors
2
3 x <- rnorm(100)
4
5 y <- rnorm(100)
6
7 # Two sample test
8
9 wilcox.test(x,y)
10 |
```

The console output shows the results of the test:

```
R 4.2.1 ~ /
> x <- rnorm(100)
> y <- rnorm(100)
> wilcox.test(x,y)

    wilcoxon rank sum test with continuity correction

data:  x and y
w = 5482, p-value = 0.2394
alternative hypothesis: true location shift is not equal to 0
```

Image showing two sample Mu test

Correlation Test:

This test is used to compare the correlation of the two vectors provided in the function call or to test for the association between paired samples.

Syntax:

```
cor.test(x,y)
```

x and y are numeric vectors.

In the below example the dataset available with dplyr package is used. If not already installed it must be installed to make use of this database.

Example:

```
# Uses mtcars dataset in R
```

```
cor.test(mtcars$mpg, mtcars$hp)
```

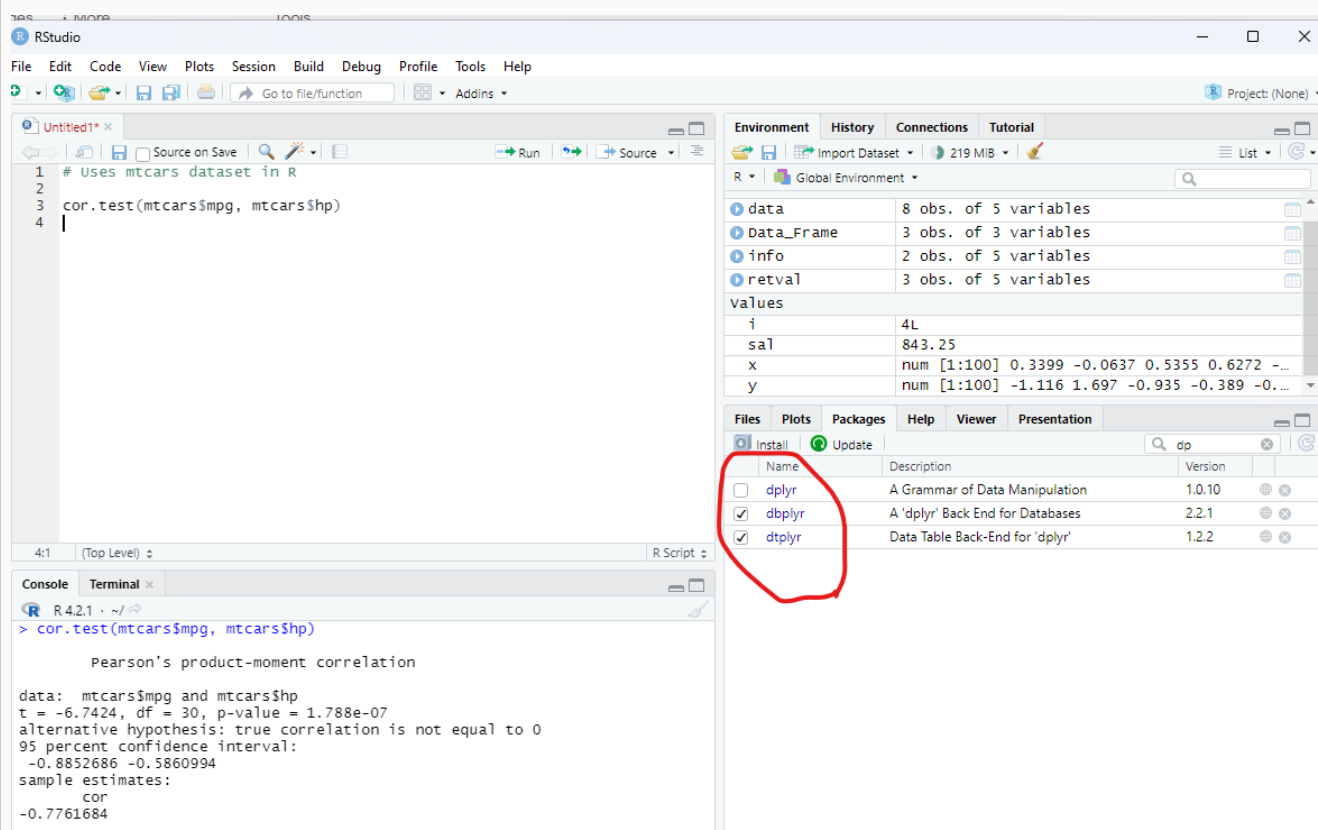


Image showing the use of correlation

Bootstrapping in R Programming:

This technique is used in inferential statistics that work on building random samples of single datasets again and again. This method allows calculating measures such as mean, median, mode, confidence intervals etc. of the sampling.

Process of bootstrapping in R language:

1. Selecting the number of bootstrap samples.
2. Select the size of each sample.
3. For each sample, if the size of the sample is less than the chosen sample, then select a random observation from the dataset and add it to the sample.
4. Measure the statistic on the sample.
5. Measure the mean of all calculated sample values.

Methods of Bootstrapping:

There are two methods of Bootstrapping:

Residual Resampling - This method is also known as model based resampling. This method assumes that the model is correct and errors are independent and distributed identically. After each resampling, variables are redefined and new variables are used to measure the new dependent variables.

Bootstrap Pairs - In this method, dependent and independent variables are used together as pairs of sampling.

Types of confidence intervals in Bootstrapping:

This type of computational value calculated on sample data in statistics. It produces a range of values or interval where the true value lies for sure. There are 5 types of confidence intervals in bootstrapping as follows:

Basic - It is also known as Reverse percentile interval and is generated using quantiles of bootstrap data distribution.

Normal confidence interval.

Stud - In studentized CI, data is normalized with centre at 0 and standard deviation 1 correcting the skew of distribution.

Perc - Percentile CI is similar to basic CI but the formula is different.

Syntax:

`boot(data, statistic, R)`

data - represents dataset

statistic - represents statistic functions to be performed on dataset.

R - represents the number of samples.

Example:

```
# Installation of the libraries required
```

```
install.packages("boot")
```

```
# Load the library
```

```
library(boot)
```

```
# Creating a function to pass into boot() function
```

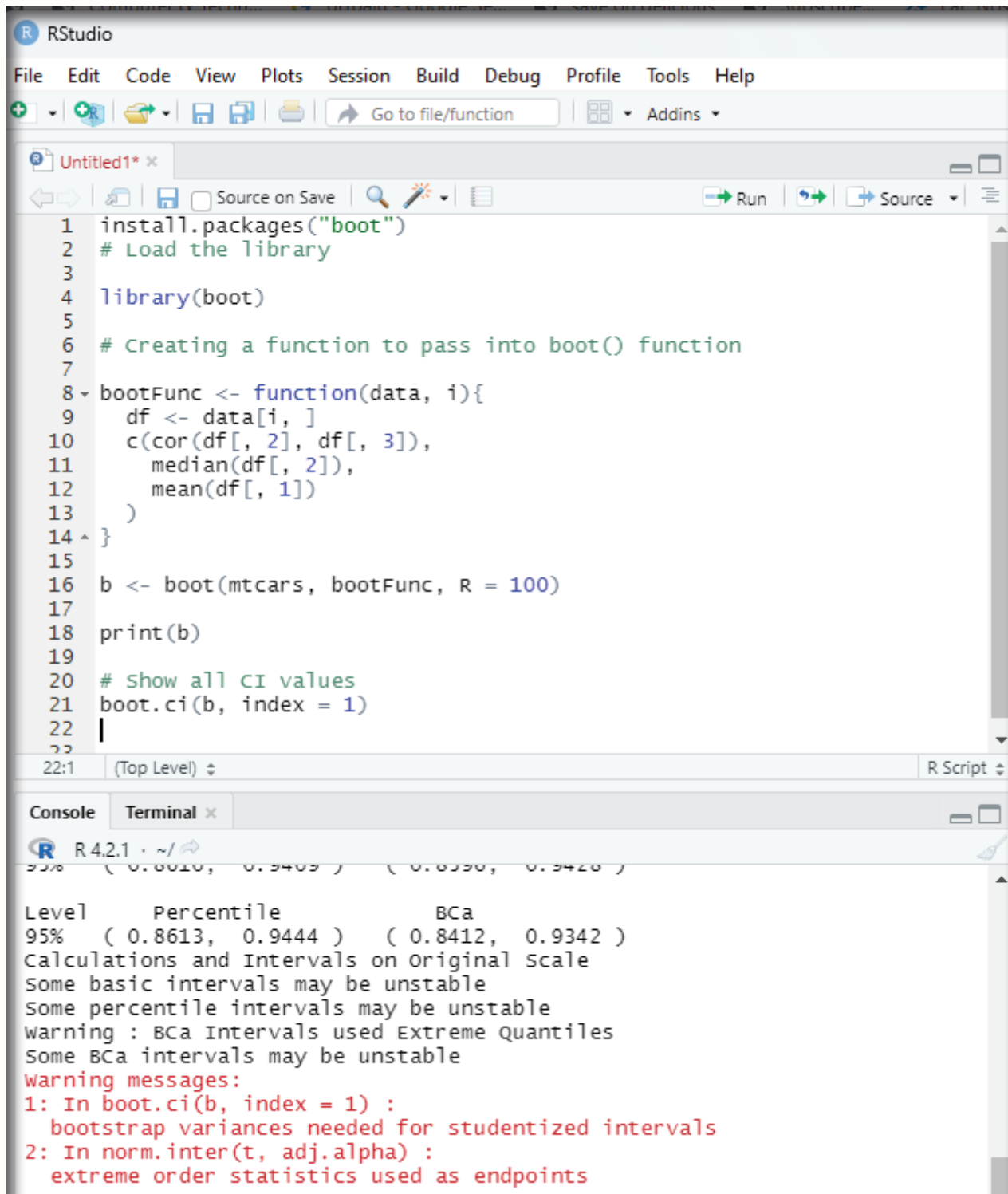
```
bootFunc <- function(data, i){  
  df <- data[i, ]  
  c(cor(df[, 2], df[, 3]),  
    median(df[, 2]),  
    mean(df[, 1])  
  )  
}
```

```
b <- boot(mtcars, bootFunc, R = 100)
```

```
print(b)
```

```
# Show all CI values
```

```
boot.ci(b, index = 1)
```

```
1 install.packages("boot")
2 # Load the library
3
4 library(boot)
5
6 # Creating a function to pass into boot() function
7
8 bootFunc <- function(data, i){
9   df <- data[i, ]
10   c(cor(df[, 2], df[, 3]),
11     median(df[, 2]),
12     mean(df[, 1])
13 )
14 }
15
16 b <- boot(mtcars, bootFunc, R = 100)
17
18 print(b)
19
20 # Show all CI values
21 boot.ci(b, index = 1)
22
23
```

22:1 (Top Level) ↕ R Script

Console Terminal x

R 4.2.1 ~ /

```
95% ( 0.8613, 0.9444 ) ( 0.8412, 0.9342 )
Level Percentile BCa
95% ( 0.8613, 0.9444 ) ( 0.8412, 0.9342 )
Calculations and Intervals on Original scale
Some basic intervals may be unstable
Some percentile intervals may be unstable
Warning : BCa Intervals used Extreme Quantiles
Some BCa intervals may be unstable
warning messages:
1: In boot.ci(b, index = 1) :
  bootstrap variances needed for studentized intervals
2: In norm.inter(t, adj.alpha) :
  extreme order statistics used as endpoints
```

Image showing Bootstrapping

Time series analysis using R:

Time Series in R is used to see how an object behaves over a period of time. This analysis can be performed using `ts()` function with some parameters. Time series takes the data vector and each data is connected with timestamp value as given by the user. This function can be used to learn and forecast the behavior of an asset during a period of time.

Syntax:

```
<- ts(data, start, end, frequency)
```

data - represents the data vector

start - represents the first observation in time series

end - represents the last observation in time series

frequency - represents number of observations per unit time. Example : frequency = 1 for monthly data.

Example:

Analysing total number of positive cases of COVID 19 on a weekly basis from 10th Jan to 30th April 2020.

Weekly data of covid positive cases between 10th Jan to 30th April 2020.

```
x <- c(690, 6000, 18000, 67342, 79231, 89432, 129876, 138721, 149842, 169826, 187421,
      192781, 208721)
```

Library need to calculate decimal_date function

```
library(lubridate)
```

creating time series object

from date `10 January, 2020

```
mts <- ts(x, start = decimal_date(ymd("2020-01-10")),
          frequency = 365.25 / 7)
```

plotting the graph

```
plot(mts, xlab = "Weekly Data",
      ylab = "Total Positive Cases",
      main = "COVID-19 Pandemic",
      col.main = "darkgreen")
```

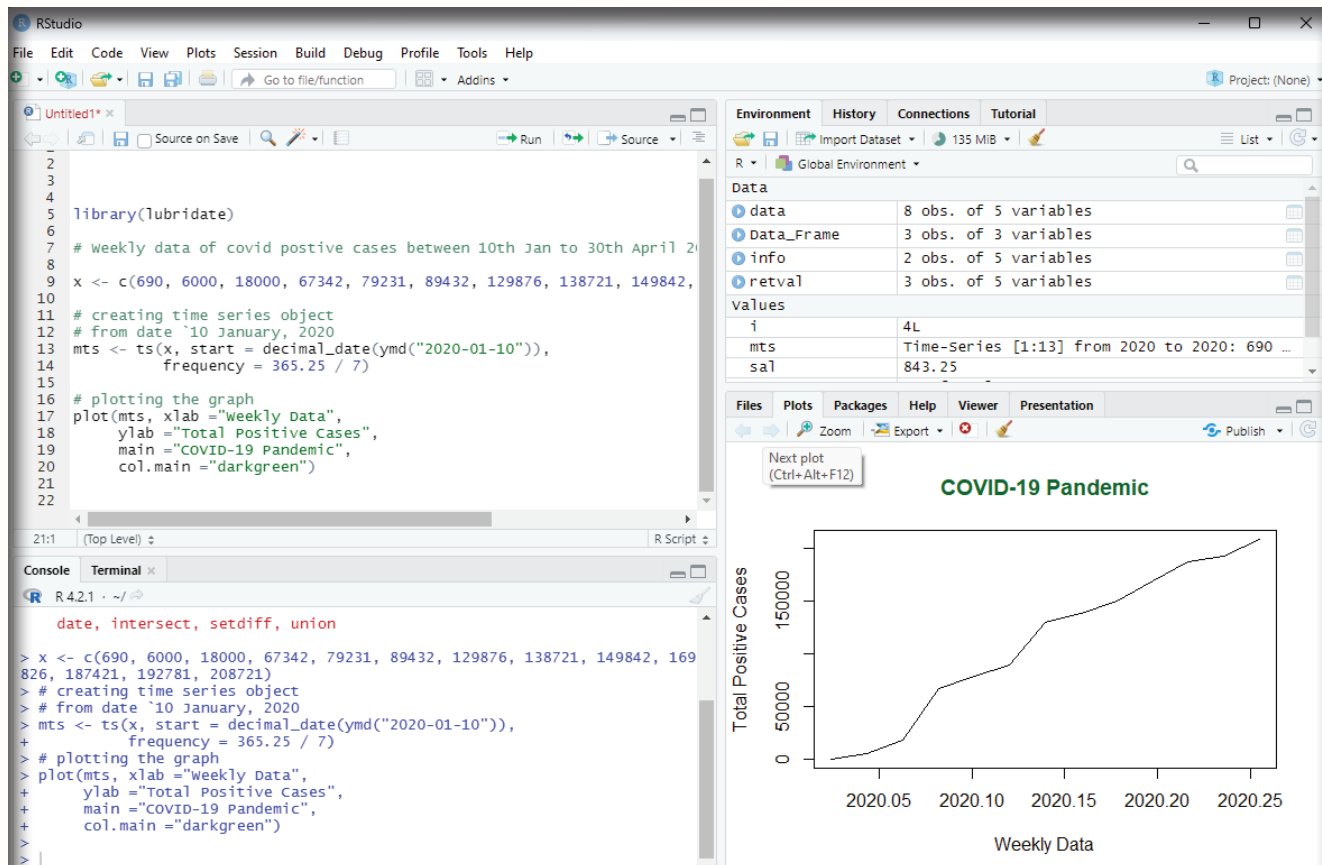


Image showing Time series analysis

Multivariate Time series:

This is used to create multiple time series in a single chart.

Weekly data of Covid positive cases

Weekly deaths from 10th Jan to 30 th April 2020

*positiveCases <- c(780, 9823, 32256, 46267,
78743, 87820, 95314, 126214,
218843, 471497, 936851,
1508725, 2072113)*

*deaths <- c(17, 270, 565, 1261, 2126, 2800,
3285, 4628, 8951, 21283, 47210,
88480, 138475)*

*# creating multivariate time series object
from date 10 January, 2020*

```
mts <- ts(cbind(positiveCases, deaths),
start = decimal_date(ymd("2020-01-22")),
frequency = 365.25 / 7)
```

```
# plotting the graph
plot(mts, xlab = "Weekly Data",
main = "COVID-19 Cases",
col.main = "darkgreen")
```

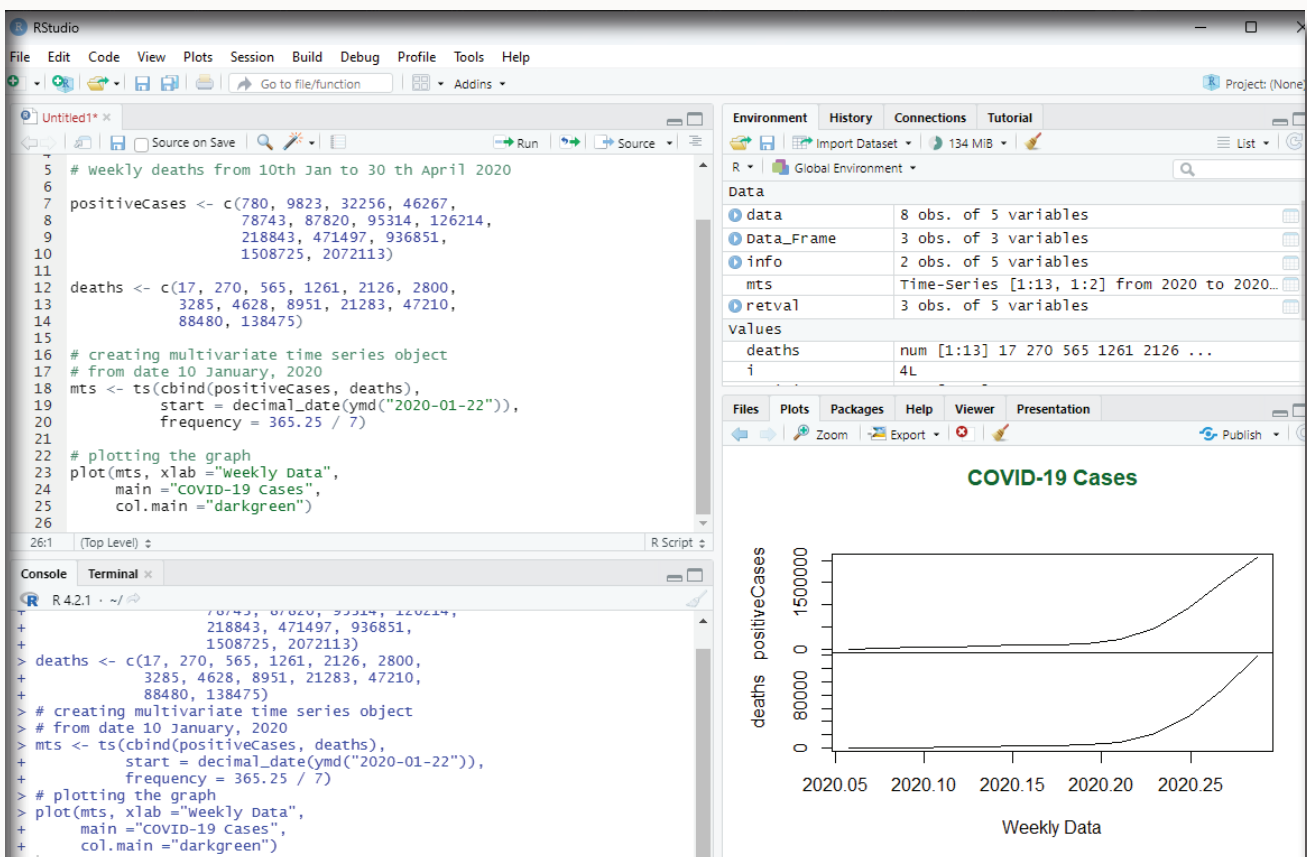


Image showing multivariate time series

Forecasting:

Forecasting can be done on time series using some models available in R. Arima automated model is commonly used.

```
# Weekly data of COVID-19 cases from  
# 22 January, 2020 to 15 April, 2020  
x <- c(580, 7813, 28266, 59287, 75700,  
87820, 95314, 126214, 218843,  
471497, 936851, 1508725, 2072113)  
  
# library required for decimal_date() function  
library(lubridate)  
  
install.packages("forecast")  
  
# library required for forecasting  
library(forecast)  
  
# output to be created as png file  
png(file = "forecastTimeSeries.png")  
  
# creating time series object  
# from date 22 January, 2020  
mts <- ts(x, start = decimal_date(ymd("2020-01-22")),  
frequency = 365.25 / 7)  
  
# forecasting model using arima model  
fit <- auto.arima(mts)  
  
# Next 5 forecasted values  
forecast(fit, 5)  
  
# plotting the graph with next  
# 5 weekly forecasted values  
plot(forecast(fit, 5), xlab = "Weekly Data",  
ylab = "Total Positive Cases",  
main = "COVID-19 Pandemic", col.main = "darkgreen")
```


Tidyverse

Though base R package includes many useful functions and data structures that can be used to accomplish a wide variety to data science tasks, the third party “tidyverse” package supports a comprehensive data science workflow. The tidyverse ecosystem includes many sub-packages designed to address specific components of the workflow. 80% of data analysis time is spent cleaning and preparing the data collected. The user should aim at creating a data standard to facilitate exploration and analysis. Tidyverse helps the user to cut down on data analysis time spent of cleaning and preparing the collected data.

Tidyverse is a coherent system of packages for importing, tidying, transforming, exploring and visualizing data. These packages are intended to make statisticians and data scientists more productive by guiding them through workflows that facilitate communication, and result in reproducible work products.

Core packages of tidyverse are:

readr - The main function of this package is to facilitate the import of file based data into a structured data format. The readr package includes seven functions for importing file-based datasets which include csv, tsv, delimited, fixed width, white space separated and web log files.

Data is imported into a structure called a tibble. Tibbles are nothing but the tidyverse implementation of a data frame. They are similar to data frames, but are basically a newer and more advanced version. There are important differences between tibbles and data frames. Tibbles never converts data types of variables. They also don't change the names of variables or create row names. Tibbles also has a refined print method that shows only the first 10 rows, and all columns that will fit the screen. Tibbles also prints the column type along with the name. Tibbles are usually considered as objects by R.

tidyr - Data tidying is a consistent way of organizing data in R. This is facilitated through tidyr package. There are three rules that one needs to follow to make a dataset tidy. Firstly, each variable should have its own column, second, each observation must have its own row, and finally each value must have its own cell.

dplyr - This package is a very important component of tidyverse. It includes 5 key functions for transforming the data in various ways. These functions include:

```
filter()
arrange()
select()
mutate()
summarize()
```

All these functions work similarly. The first argument is the data frame the user is operating on, the next N number of arguments are the variables to include. The results of calling all 5 functions is the creation of a new data frame that is a transformed version of the data frame passed to the function.

ggplot2 - This package is a data visualization package for R. It is an implementation of the Grammar of Graphics which include data, aesthetic mapping, geometric objects, statistical transformations, scales, coordinate systems, position adjustments and faceting.

Using ggplot2 one can create many forms of charts, graphs including bar charts, box plots, violin plots, scatter plots, regression lines and more. This package offers a number of advantages when compared to other visualization techniques available in R. They include a consistent style for defining the graphics, a high level of abstraction for specifying plots, flexibility, a built-in theming system for plot appearance, mature and complete graphics system and access to many ggplot2 users for support.

Other tidyverse ecosystem includes a number of other supporting packages including stringr, purr, forcats and others.

Installation of tidyverse:

This can be done by typing the following command in the scripting window.

```
install.packages("tidyverse")
```

Another way of installing packages:

Packages pane is located in the lower right portion of RStudio window. In order to install a new package using this pane, the install button should be clicked. In the packages textbox tidyverse which is the name of package that needs to be installed is typed. The user should ensure that install dependencies box is checked before clicking the install button. The install process will start as soon as the user clicks the install button.

Attaching tidyverse library and packages: This library along with tibble package that contains sample database should be attached to the R environment. This can be done by selecting and opening the Packages tab in the lower right portion of RStudio window. From the packages list tidyverse and tibble are chosen to be attached by placing a check in the check box in front of them.

To import a test database contained as tibble table the following code is used in the scripting window.

```
as_tibble(iris)
```

Tibble displays only 10 rows and column that fits into the screen.

Even though it displays only ten rows and the number of columns that could fit into the window the total number of rows and columns present in the data set is revealed. Above every column the following details can be seen:

```
<dbl> - double  
<dbl> - double  
<dbl> - double  
<dbl> - double
```


<fct> - factor

Before embarking on cleaning up the data set the user should know the common problems with messy data-sets:

1. Column headers are values, not variable names.
2. Observations are scattered across rows.
3. Variables are stored both in rows and columns.
4. Multiple variables are stored in one column.
5. Multiple types of observational units are stored in the same table.
6. A single observational unit is stored in multiple tables.

In the scripting window key in the following code:

table4a

On clicking the run button a table as shown below will be displayed in the console window.

```
>table4a
# A tibble: 3 × 3
  country `1999` `2000`
* <chr> <int> <int>
1 Afghanistan 745 2666
2 Brazil 37737 80488
3 China 212258 213766
> table
```

The first line indicates the title of the table.

Next to the comment # sign is displayed the details of the tibble (table with 3x3 dimensions, 3 rows and 3 columns).

This tibble has one column for country and one column each for the year 1999 and 2000 as shown above. The column 1999 and column 2000 headers are actually values of the variable year. Under the country column the following countries are listed along with observations for the year 1999 and 2000 respectively. The countries listed in the country column are Afghanistan, Brazil and China. These columns should be pivotted in to rows in order to make meaningful analysis of the dataset.

Code that is used to pivot columns into rows:

```
pivot_longer( table4a, c('1999', '2000'),
names_to = 'year', values_to = 'cases')
```

Output as displayed in the console window:

```
country year cases
```

<chr> <chr> <int>

- 1 Afghanistan 1999 745
- 2 Afghanistan 2000 2666
- 3 Brazil 1999 37737
- 4 Brazil 2000 80488
- 5 China 1999 212258
- 6 China 2000 213766

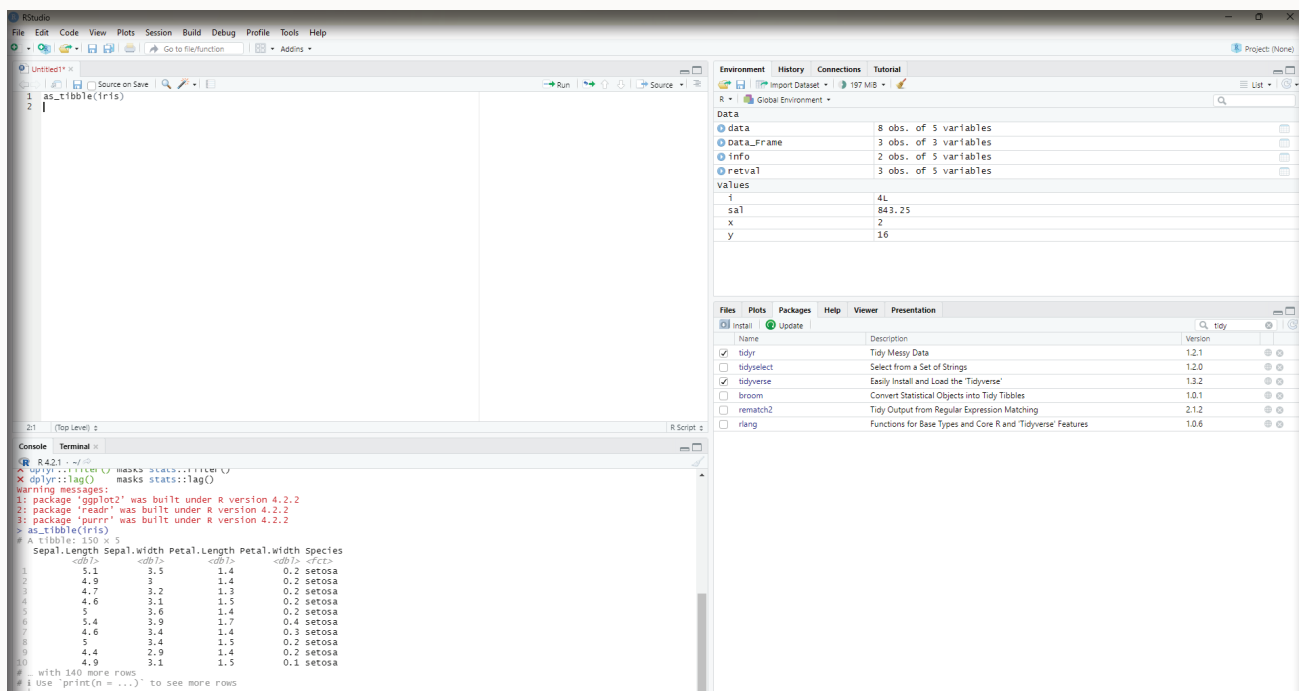


Image showing iris data base displayed as a tibble. Note the package tidy has been enabled in the package screen.

```
> table4a
# A tibble: 3 x 3
  country    `1999` `2000`
  <chr>      <int> <int>
1 Afghanistan    745    2666
2 Brazil        37737  80488
3 china         212258 213766
> |
```

Table 4a on display

```
# A tibble: 6 × 3
  country    year  cases
  <chr>      <chr> <int>
1 Afghanistan 1999     745
2 Afghanistan 2000    2666
3 Brazil       1999   37737
4 Brazil       2000   80488
5 China        1999  212258
6 china        2000  213766
```

Image showing the result of pivot_longer() function

If the following code is typed into the scripting window and run a table will open up in the console window.

Code:

table2

Output:

```
# A tibble: 12 × 4
  country year type count
  <chr> <int> <chr> <int>
1 Afghanistan 1999 cases 745
2 Afghanistan 1999 population 19987071
3 Afghanistan 2000 cases 2666
4 Afghanistan 2000 population 20595360
5 Brazil 1999 cases 37737
6 Brazil 1999 population 172006362
7 Brazil 2000 cases 80488
8 Brazil 2000 population 174504898
9 China 1999 cases 212258
10 China 1999 population 1272915272
11 China 2000 cases 213766
12 China 2000 population 1280428583
```

	country	year	type	count
	<chr>	<int>	<chr>	<int>
1	Afghanistan	1999	cases	745
2	Afghanistan	1999	population	19987071
3	Afghanistan	2000	cases	2666
4	Afghanistan	2000	population	20595360
5	Brazil	1999	cases	37737
6	Brazil	1999	population	172006362
7	Brazil	2000	cases	80488
8	Brazil	2000	population	174504898
9	China	1999	cases	212258
10	China	1999	population	1272915272
11	China	2000	cases	213766
12	China	2000	population	1280428583

Image showing the result of table2 command in the scripting window

Observations are spread across rows. One observation is spread across two rows. One can note that there are two entries for 1999 as far as Afghanistan is concerned. The same scenario is observed for other countries also. Data needs to be pivot the data wider.

Code for pivoting the data wider:

```
pivot_wider( table2,  
names_from = 'type', values_from = count)
```

output:

A tibble: 6 × 4

```
country year cases population
<chr> <int> <int> <int>
1 Afghanistan 1999 745 19987071
2 Afghanistan 2000 2666 20595360
3 Brazil 1999 37737 172006362
4 Brazil 2000 80488 174504898
5 China 1999 212258 1272915272
6 China 2000 213766 1280428583
```

Pivot wider and pivot longer are otherwise called as spread and gather.

Tidyverse has other tools for importing data of various formats and manipulating the same.

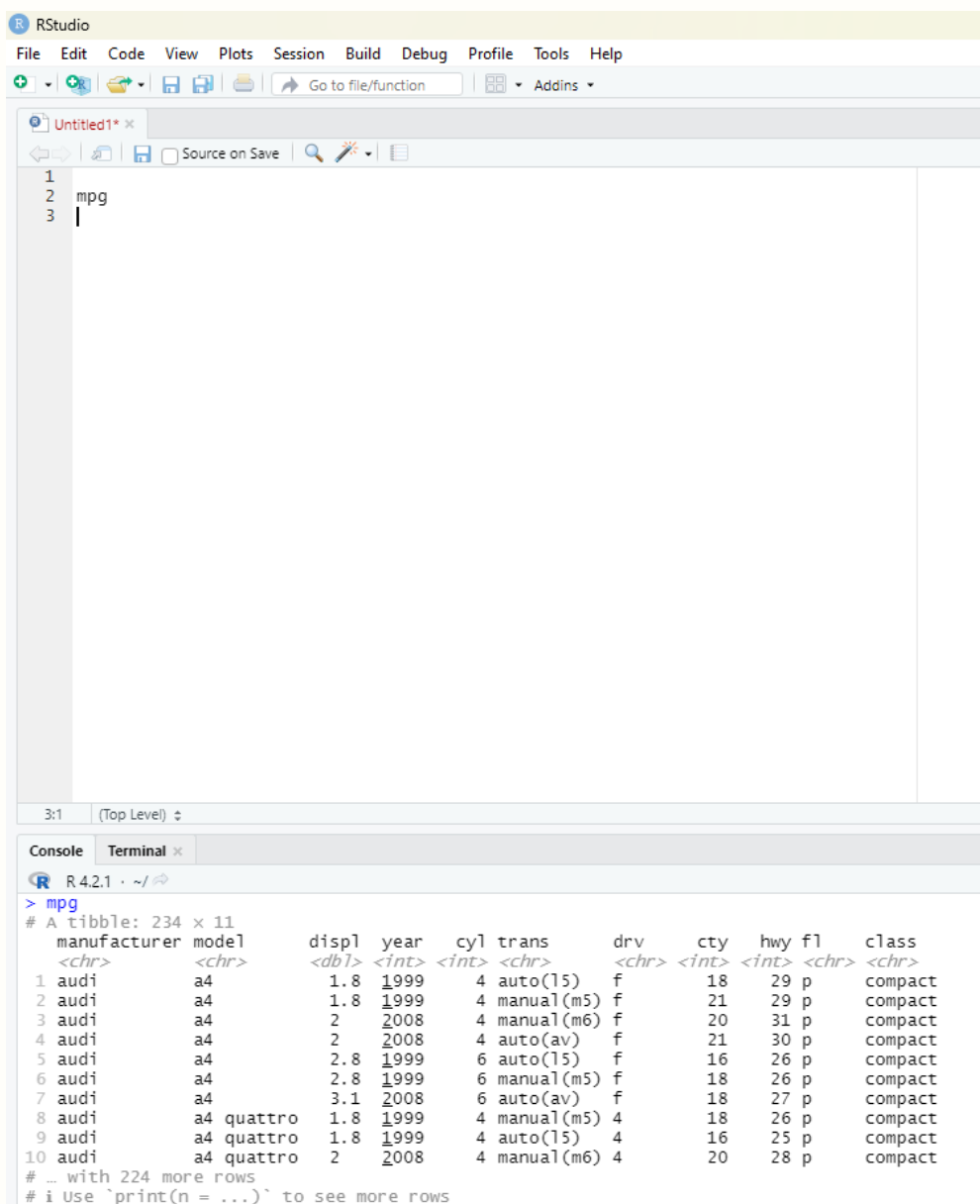
Tools that take tidy datasets as input and return tidy datasets as output.

Pipe operator is another tool in tidyverse that is really useful.

%>% the pipe operator.

Default behavior of pipe operator is to place the left hand side as the first argument for the function on the right side.

If the user keys in *mpg* and executes the code in scripting window a data frame would open in the console window. This data frame contains observations collected by US environmental protection agency in 38 models of car.



```
RStudio
File Edit Code View Plots Session Build Debug Profile Tools Help
Go to file/function
Addins
Untitled1*
1
2 mpg
3 |

3:1 (Top Level)
Console Terminal
R 4.2.1 ~ /
> mpg
# A tibble: 234 x 11
  manufacturer model displ year cyl trans drv cty hwy fl class
  <chr> <chr> <dbl> <int> <int> <chr> <chr> <int> <int> <chr> <chr>
1 audi a4 1.8 1999 4 auto(l5) f 18 29 p compact
2 audi a4 1.8 1999 4 manual(m5) f 21 29 p compact
3 audi a4 2 2008 4 manual(m6) f 20 31 p compact
4 audi a4 2 2008 4 auto(av) f 21 30 p compact
5 audi a4 2.8 1999 6 auto(l5) f 16 26 p compact
6 audi a4 2.8 1999 6 manual(m5) f 18 26 p compact
7 audi a4 3.1 2008 6 auto(av) f 18 27 p compact
8 audi a4 quattro 1.8 1999 4 manual(m5) 4 18 26 p compact
9 audi a4 quattro 1.8 1999 4 auto(l5) 4 16 25 p compact
10 audi a4 quattro 2 2008 4 manual(m6) 4 20 28 p compact
# ... with 224 more rows
# Use `print(n = ...)` to see more rows
```

Image showing the result of keying mpg in scripting window

Output:

```
# A tibble: 234 × 11
  manufacturer model displ year cyl trans drv cty hwy fl class
  <chr> <chr> <dbl> <int> <int> <chr> <chr> <int> <int> <chr> <chr>
1 audi a4 1.8 1999 4 auto(l5) f 18 29 p comp...
2 audi a4 1.8 1999 4 manual(m... f 21 29 p comp...
3 audi a4 2 2008 4 manual(m... f 20 31 p comp...
4 audi a4 2 2008 4 auto(av) f 21 30 p comp...
5 audi a4 2.8 1999 6 auto(l5) f 16 26 p comp...
6 audi a4 2.8 1999 6 manual(m... f 18 26 p comp...
7 audi a4 3.1 2008 6 auto(av) f 18 27 p comp...
8 audi a4 quattro 1.8 1999 4 manual(m... 4 18 26 p comp...
9 audi a4 quattro 1.8 1999 4 auto(l5) 4 16 25 p comp...
10 audi a4 quattro 2 2008 4 manual(m... 4 20 28 p comp...
# ... with 224 more rows
# i Use `print(n = ...)` to see more rows
```

In the above database (which is also known as tibble in tidyverse) only 10 rows are visible. The number of columns are restricted by the screen space. A command “print(n=...)” is used to see more rows.

Among the variables in mpg are:

1. displ - car’s engine size in litres
2. hwy - car’s fuel efficiency on the highway. A car with low fuel efficiency consumes more fuel than a car with high fuel efficiency when they travel the same distance.

Creating a ggplot:

In order to plot mpg the following code should be run.

Code:

```
ggplot(data=mpg)+
```

```
  geom_point(mapping = aes(x = displ, y = hwy))
```

This plot clearly shows a negative relationship between engine size (displ) and fuel efficiency (hwy). Cars with big engines use more fuel.

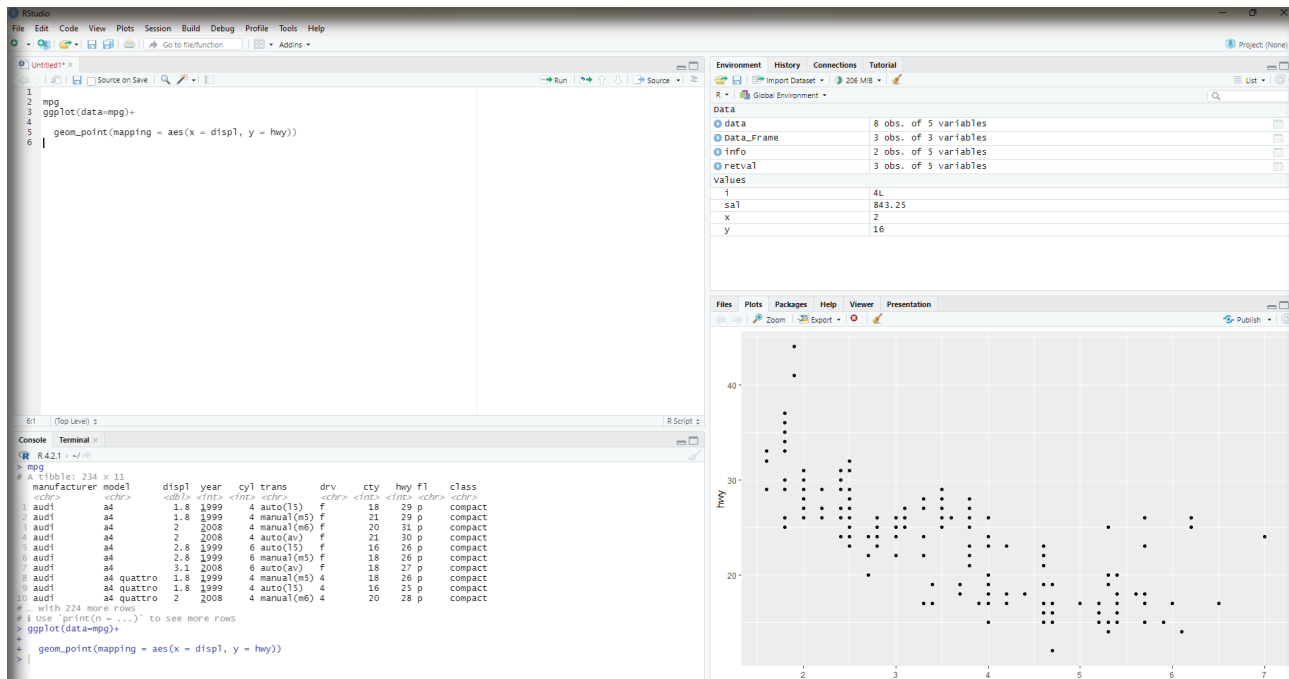


Image showing ggplot being used to create graphs

Aesthetic mappings:

Great value of a picture is that it forces the viewer to notice what was not expected.

In the scatter plot created from the database mpg one can see a group of points that are outside of the linear trend indicating that these cars demonstrated a higher mileage than what is expected. How can these outliers be explained? One hypothesis could be that these cars could be hybrid variety. The tibble titled mpg has a variable titled class. The class variable classifies car into groups such as compact, mid-size and SUV. The user can add a third variable class, to a two dimensional scatter plot by mapping it to an aesthetic. Aesthetic is described as a visual property of the objects in the plot. One can display a point in different ways by changing the values of its aesthetic properties.

Aesthetics including the following parameters:

1. Size
2. Shape
3. Color of the points

Information about the data can be conveyed by mapping the aesthetics in the plot to the variables in the dataset. In this example one can map the colors of the points to the class variable to reveal the type of each car. In order to map an aesthetic to a variable, the name of the aesthetic is associated to the name of the variable inside aes(). ggplot2 will automatically assign a unique level of the aesthetic by assigning it an unique color. This process is known as scaling. ggplot2 will also add a legend that explains the levels corresponding to the

values.

Code for using aesthetics:

```
ggplot(mpg, aes(displ, hwy, colour = class)) +  
  geom_point()
```

Common errors in R coding:

R is extremely fussy about code syntax. A misplaced character can be a cause of problems. The user should make sure that every (is matched with a) and every " is paired with another ". Sometimes when the code is run from the scripting window if nothing happens in the console window lookout for the + sign. If it is displayed it indicates the expression is incomplete and R is waiting for the user to complete it.

One other common problem that can occur during creation of ggplot2 graphics is to put the + in the wrong place: it has to occur at the end of the line, not at the beginning.

In R help is around the corner. Help can be assessed by running ? function name in the console, or selecting the function name and pressing F1 in R studio.

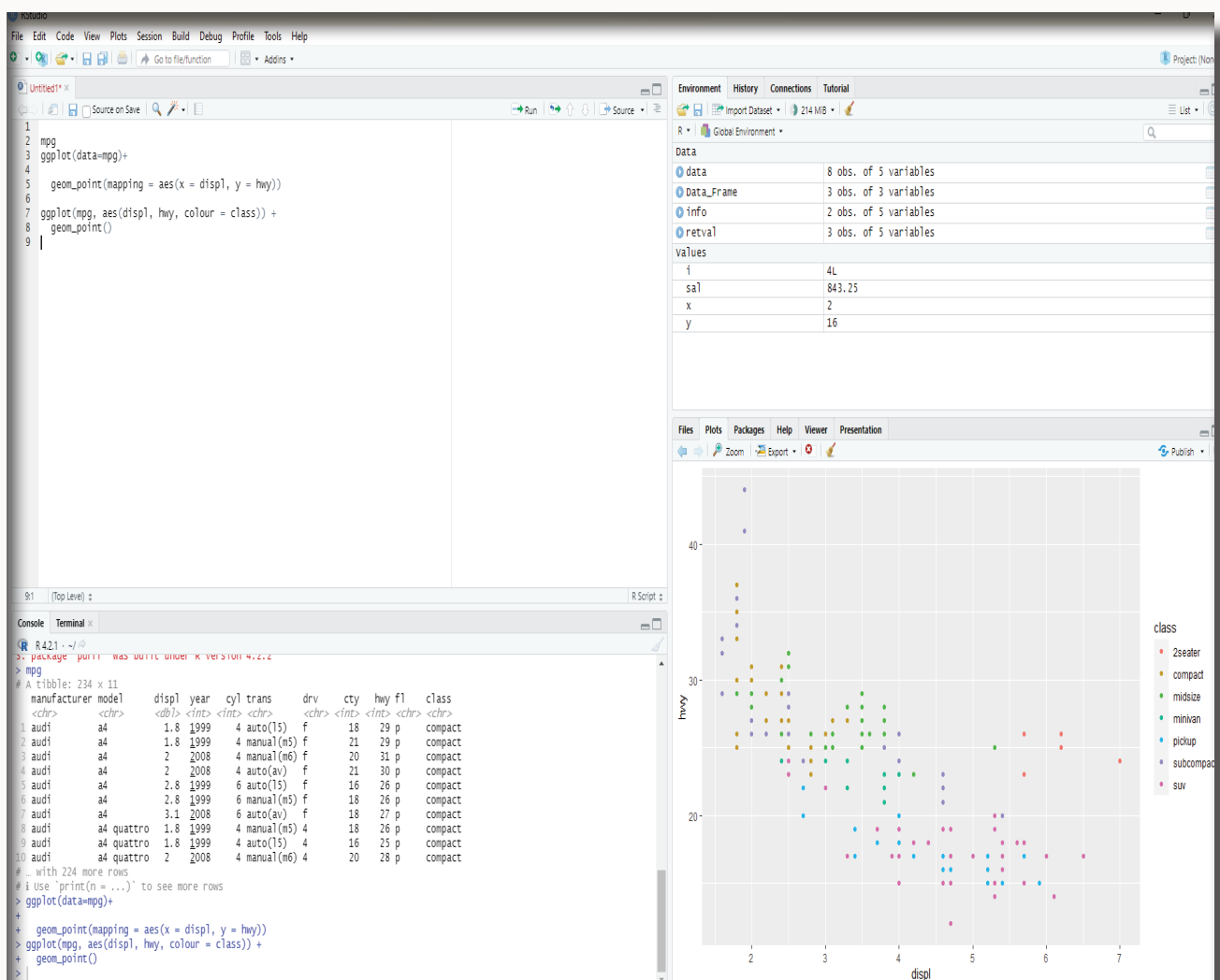


Image showing the effects of aesthetics code

Facets:

One way of adding additional variables is with aesthetics. Another useful way for adding categorical variables is to split the plot into facets, subplots that each display one subset of the data.

In order to facet the plot by a single variable, the `facet_wrap()` function is used. The first argument of the `facet_wrap()` should be a formula, which is created with `~` followed by a variable name. (Formula is the name of the data structure in R and not a synonym for equation). The variable that is passed to the `facet_wrap()` should be discrete.

Code:

```
ggplot(data=mpg)+  
geom_point(mapping=aes(x=displ, y=hwy))+  
facet_wrap(~ class, nrow=2)
```

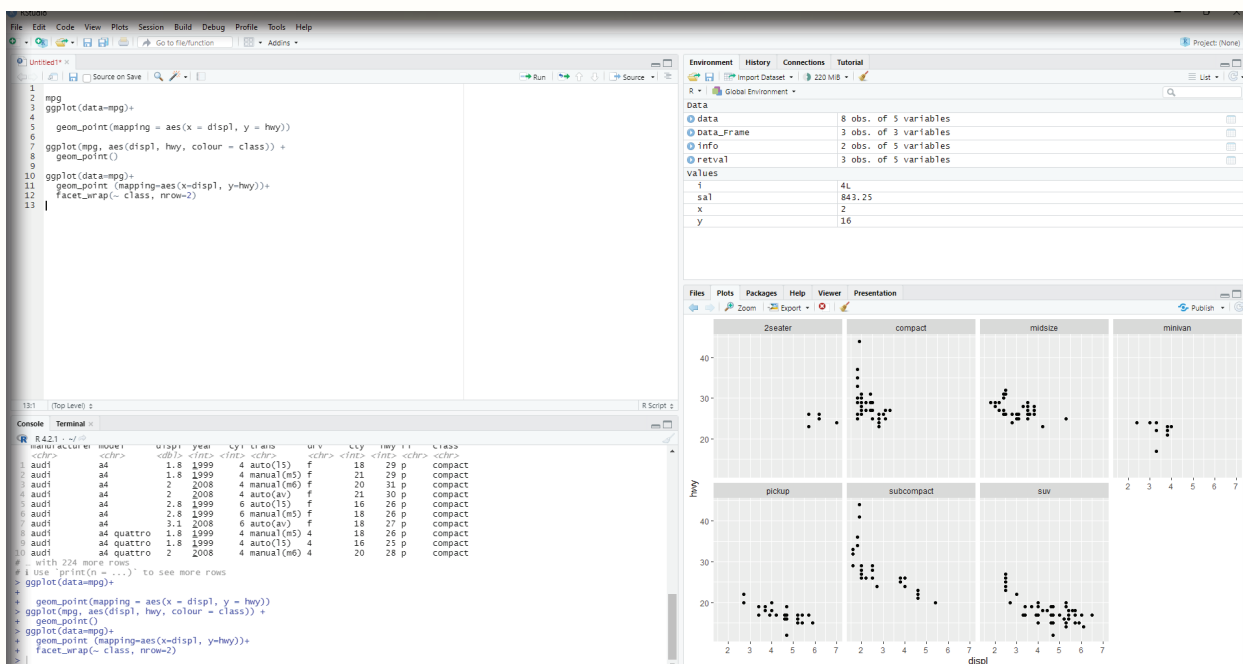


Image showing the result of Facets code

In order to facet the plot on the combination of two variables, `facet_grid()` is added to the plot call. The first argument of `facet_grid()` is also a formula. The formula this time should contain two variable names separated by a `~`.

```
ggplot(data=mpg) +
  geom_point(mapping = aes(x=displ, y=hwy))+
  facet_grid (drv~cyl)
```

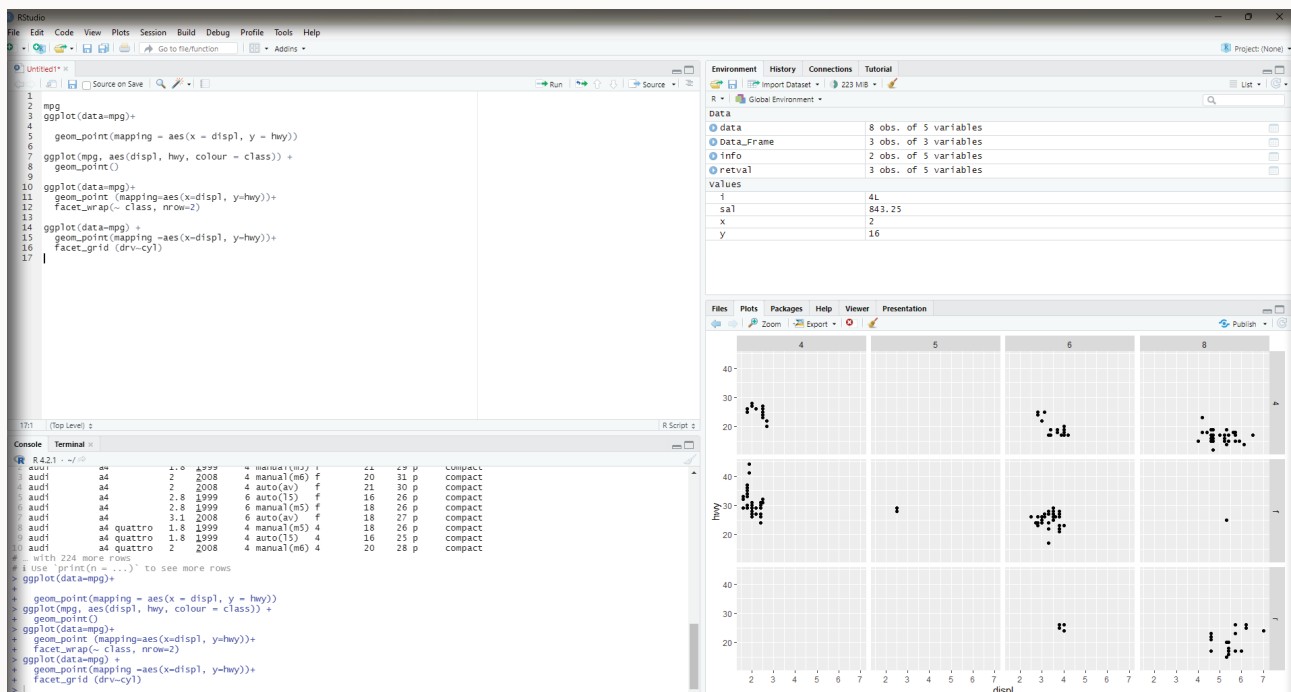


Image showing facet grid

Geom:

A geom is the geometrical object that a plot uses to represent data. People often describe plots by the type of geom that the plot uses. Bar charts use bar geoms, line charts use line geoms, boxplots use boxplot geoms and so on. On the other hand scatterplots use the point geom. Different geoms can be used to plot the same data.

To change the geom in the plot, the geom function is added to ggplot().

#left

*ggplot(data=mpg)+
geom_point(mapping =aes(x=displ,y=hwy))*

right

*ggplot (data=mpg)+
geom_smooth(mapping=aes(x=displ, y=hwy))*

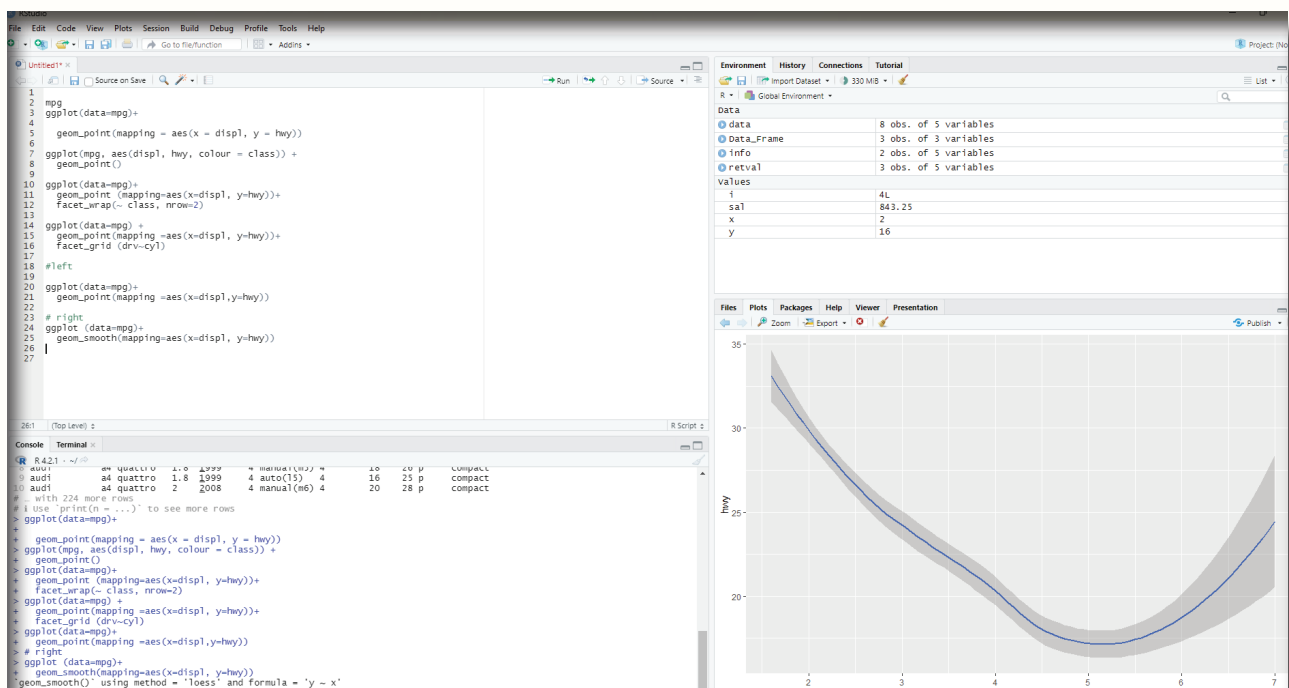


Image showing the use of Geom

Statistical transformations:

Bar charts could appear simple. They could reveal some details that could be interesting to the user. In the example below the chart displays total number of diamonds in the diamonds dataset, grouped by cut. This dataset comes along with ggplot2 package. The user should ensure that ggplot2 package is selected by ticking the box in-front of the package name in the packages window.

This dataset contains information of about 54,000 diamonds which include price, carat, color, clarity and cut for each diamond. The bar chart shows that more diamonds are available with high quality cuts than with low quality cuts.

Code:

```
ggplot(data=diamonds)+  
geom_bar(mapping=aes(x=cut))
```

On the x-axis the chart displays cut, a variable from diamonds. On the y-axis, it displays count. It should be pointed out that count is not a variable in diamonds.

1. Bar charts, histograms and frequency polygons bin the data and plot bin counts, the number of points that fall in each bin.
2. Smoothers fit a model to the data and then plot predictions from the model
3. Box plot compute a robust summary of the distribution and then displays them in a specially formatted box.

The algorithm used to calculate new values for a graph is called a stat. (Short form for statistical transformation).

The term geoms and stats can be used interchangeably.

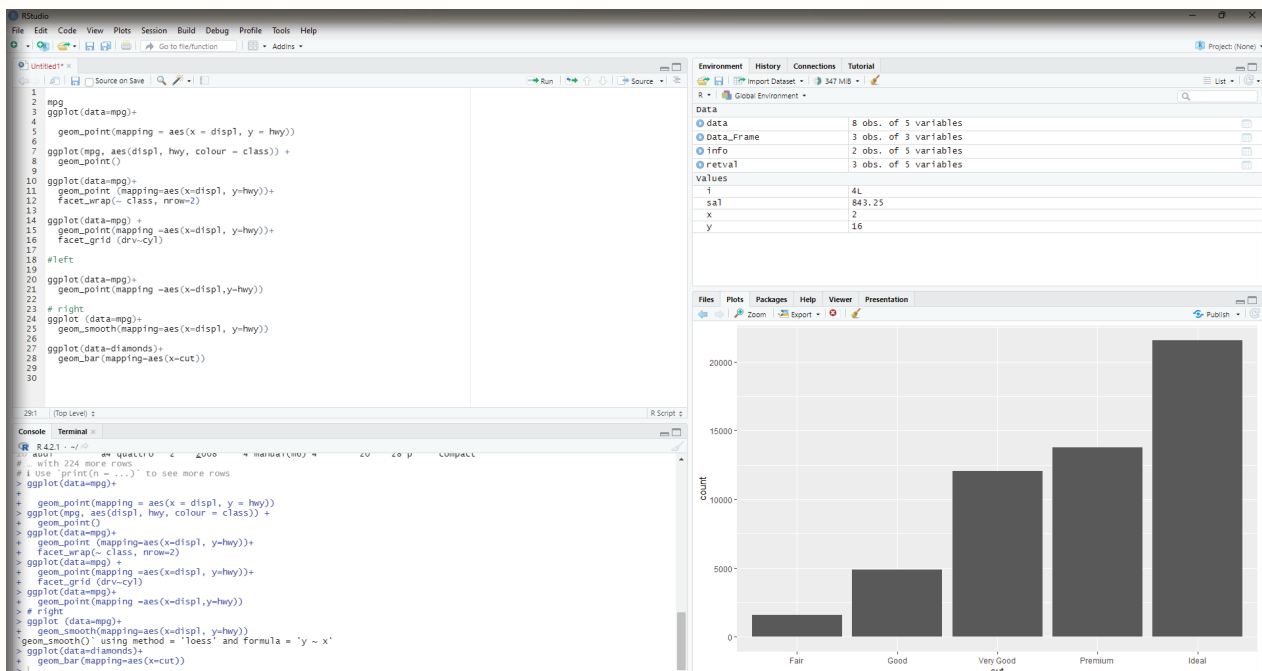


Image showing creation of bar charts

Code:

```
ggplot(data= diamonds)+  
stat_count(mapping =aes(x=cut))
```

This code works because every geom has a default stat; and every stat has a default geom. Geoms can be used without worrying about the underlying statistical formation.

If the intention is to override the default mapping from transformed variables to aesthetics like display a bar chart of proportion rather than the count then the following code need to be used.

```
ggplot(data = diamonds) +  
geom_bar(mapping = aes(x = cut, y = stat(prop), group = 1))
```

One can summarize the y values for each unique x value.
The following code should be used:

```
ggplot(data = diamonds) +  
stat_summary(  
mapping = aes(x = cut, y = depth),  
fun.min = min,  
fun.max = max,  
fun = median  
)
```

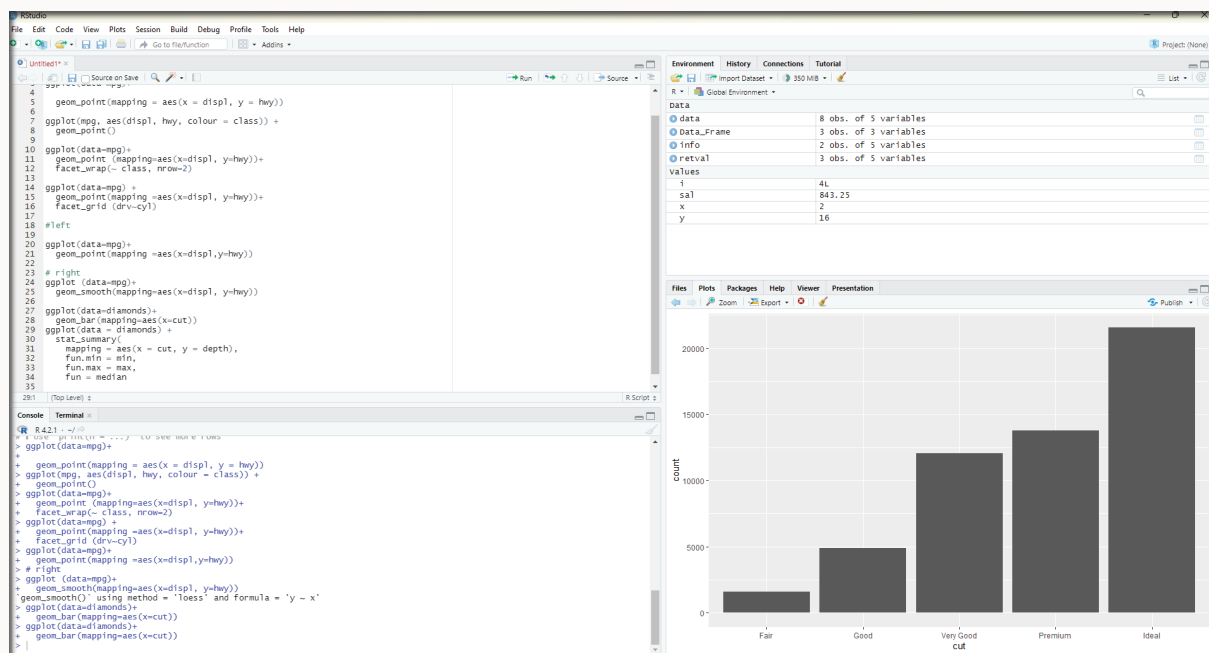


Image showing data summary as demonstrated by bar chart

Position adjustments:

One can color a bar chart using a color aesthetic or fill. Of these two fill is ideal.

*ggplot(data = diamonds) +
geom_bar(mapping = aes(x = cut, colour = cut))
ggplot(data = diamonds) +
geom_bar(mapping = aes(x = cut, fill = cut))*

Clarity can be used to stack the bars automatically. Each colored rectangle represents a combination of cut and clarity.

*ggplot(data = diamonds) +
geom_bar(mapping = aes(x = cut, fill = clarity))*

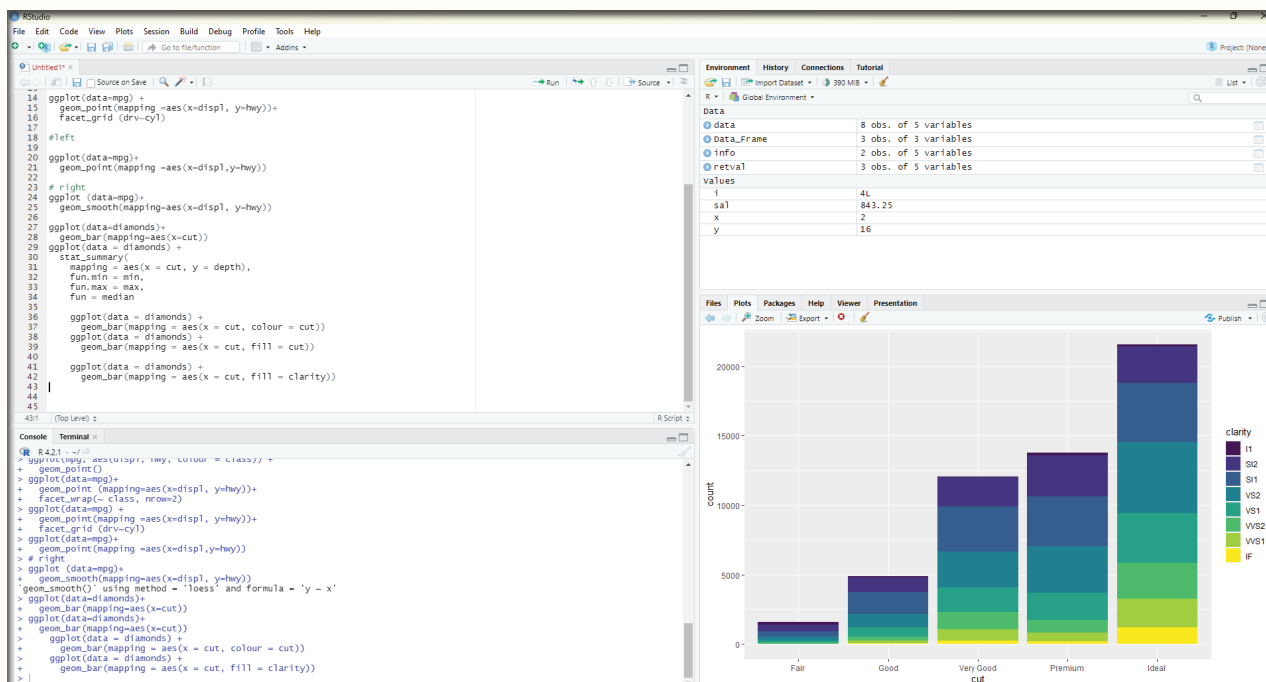


Image showing colors added to bar chart

The stacking is performed automatically by the position adjustment specified by the position argument. If the user does not desire stacked bar chart then one of these three options can be used:

identity - This will place each object exactly where it falls in the context of the graph. This may not be useful for bars, because it overlaps them. In order to see the overlapping one should make the bars slightly transparent by setting alpha to a small value or use a completely transparent setting fill = NA.

dodge - This places overlapping objects directly beside one another. This makes it easier to compare individual values.

Code:

```

ggplot(data = diamonds) +
  geom_bar(mapping = aes(x = cut, fill = clarity), position = "dodge")

```

fill - works like stacking. It makes each set of stacked bars the same height. This makes it easier to compare proportions across groups.

Code:

```
ggplot(data = diamonds) +  
  geom_bar(mapping = aes(x = cut, fill = clarity), position = "fill")
```

Another type of adjustment that would be useful for scatter plots and not in bar charts. It should be noted that not all observations can be plotted inside the graph. The values of the variables hwy and displ are rounded so the points appear on the grid and many point overlap each other. This problem goes by the term Over plotting. This makes it difficult to see where the mass of the data is. The user can avoid this over plotting by setting the position adjustment to “jitter”. position=”jitter”. This function adds a small amount of random noise to each point. This results in points being spread out and no two points are likely to receive the same amount of random noise.

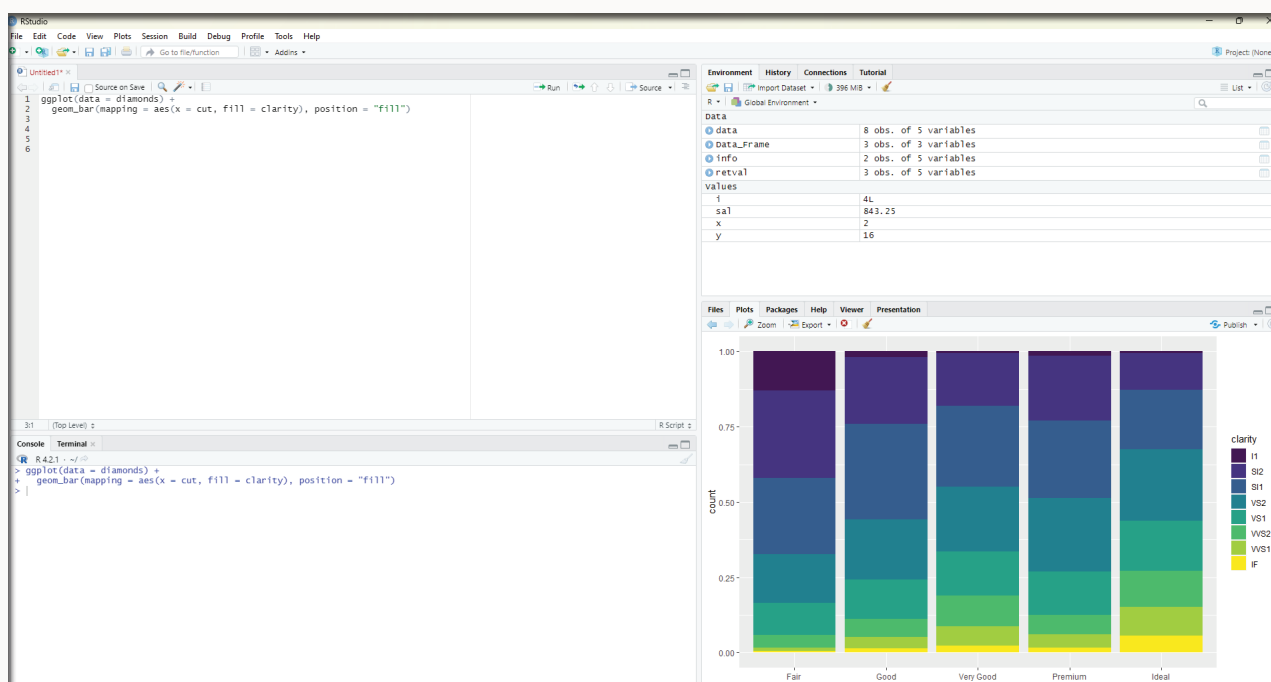


Image showing the use of fill function

Another type of adjustment that would be useful for scatter plots and not in bar charts. It should be noted that not all observations can be plotted inside the graph. The values of the variables hwy and displ are rounded so the points appear on the grid and many point overlap each other. This problem goes by the term Over plotting. This makes it difficult to see where the mass of the data is. The user can avoid this over plotting by setting the position adjustment to “jitter”. position=”jitter”. This function adds a small amount of random noise to each point. This results in points being spread out and no two points are likely to receive the same amount of random noise.

Code:

```
ggplot(data = mpg) +  
  geom_point(mapping = aes(x = displ, y = hwy), position = "jitter")
```

Adding randomness to the plot is a strange way of improving the accuracy of the graph. The graph could be less accurate at small scales.

dplyr Package in R:

This package provides tools for data manipulation in R. The dplyr package is part of the tidyverse environment. dplyr can be installed using the package installer within Rstudio. This package needs to be enabled by placing a tick inside the box in front of the package name in the packages environment of RStudio.

Alternate ways of installing and enabling dplyr package:

The following commands can be used in the scripting window and executed.

```
# command for installing the package
```

```
install.packages("dplyr")
```

```
# command for loading the package.
```

```
library("dplyr")
```

This package performs the steps in data analysis in a quicker and easy fashion.

1. It limits the choices, thereby focussing
2. There are uncomplicated “verbs”, functions present for tackling every common data manipulation and the thoughts can be translated into codes faster.
3. There are valuable back ends and hence waiting time for computer reduces.

Various functions provided by dplyr package include: 5 functions.

filter() function: Used for choosing cases and using their values as a base for doing so.

mutate() function: creates new variables.

select() function: Picks columns by name.

summarise() function: calculates summary statistics

arrange() function: Sorts the rows.

In dplyr the syntax of all the functions are very similar and they all work in a coherent manner. If the user masters these 5 functions, it will be easy for them to handle any data wrangling task. It should be remembered that data wrangling tasks should be performed one at a time.

Loading the data:

This is the first step in any data analysis. There are many example datasets available in R package. In this example diamonds dataset which is built into ggplot package is used. The first dplyr function filter() will be used.

Loading required libraries.

```
library(dplyr)
library(ggplot)
diamonds
```

If ggplot is not loaded then it should be installed from the package installer.

The database could be seen open in the console window of RStudio.

Code:

```
filter(diamonds,cut=='ideal')
```

This command filters and displays the list of diamonds under the ideal cut category.

In dplyr the function “filter” takes 2 arguments:

1. The dataframe the user is operating on
2. A conditinal expression that evaluates to TRUE or FALSE.

In the above example, diamonds has been specified as the dataframe, and cut=='ideal' as the conditional expression. For each row in the data frame, dplyr has checked whether the column cut was set to 'ideal', and returned only those rows where cut==ideal evaluted to true.

Other relational operators that can be used to compare values:

== (Equal to)

!= (Not equal to)

< (Less than)

<= (Less than or equal to)

> (Greater than)

>= (Greater than or equal to)

Note: Always use == sign to indicate equal to as single = sign is used along with assignment operator.

dplyr can also make use of the following logical operators to string together multiple different conditions in a single dplyr filter call.

! (logical not)

& (Logical and)

| (Logical or)

There are also two additional operators that could be useful when working with dplyr to filter:

%in% (Checks if a value is in an array of multiple values)

is.na (Checks whether the value is NA)

By default, dplyr performs the operations ordered and then prints the result to the screen. If the user prefers to store the result in a variable then it can be assigned as follows:

```
e_diamonds <- filter(diamonds, color == "E")
```

```
E_diamonds
```

If the user wants to overwrite the dataset (assign the result back to the diamonds dataframe) and if the user does not want to retain the unfiltered data. If the user wants to keep the original dataset then this result can be stored in e_diamonds.

Filtering Numeric values:

Numeric values are quantitative variables in a dataset. In the diamonds dataset, this includes the following variables:

Carat

Price

While working with numeric variables, it is easy to filter based on ranges of values. For example, if the user desires to get any diamonds priced between 1000 and 1500, then it can easily be filtered.

Code:

```
filter(diamonds, price >= 1000 & price <= 1500)
```

```
filter(diamonds, price >= 1500)
```

It is not advisable to use == when working with numerical variables unless the data consists of integers only and no decimals.

Anova

ANOVA also known as Analysis of Variance is a statistical test used to determine whether two or more population means are different. Simply put it is used to compare two or more groups to see if they are significantly different.

Student t-test is used to compare 2 groups, while Anova is used to compare 3 or more groups. There are several versions of ANOVA (one-way Anova, two-way ANOVA, mixed ANOVA, repeated measures ANOVA etc.

ANOVA not only compares the “between” variance (variance between the different groups) but also the variance within each group. If the between variance is significantly larger than the within variance, the group means are declared to be different.

In this chapter author will be using penguins dataset. This dataset is available in palmerpenguins package which needs to be installed first.

```
# installing palmerpenguins package
```

```
install.packages("palmerpenguins")
```

```
# Calling the dataset
```

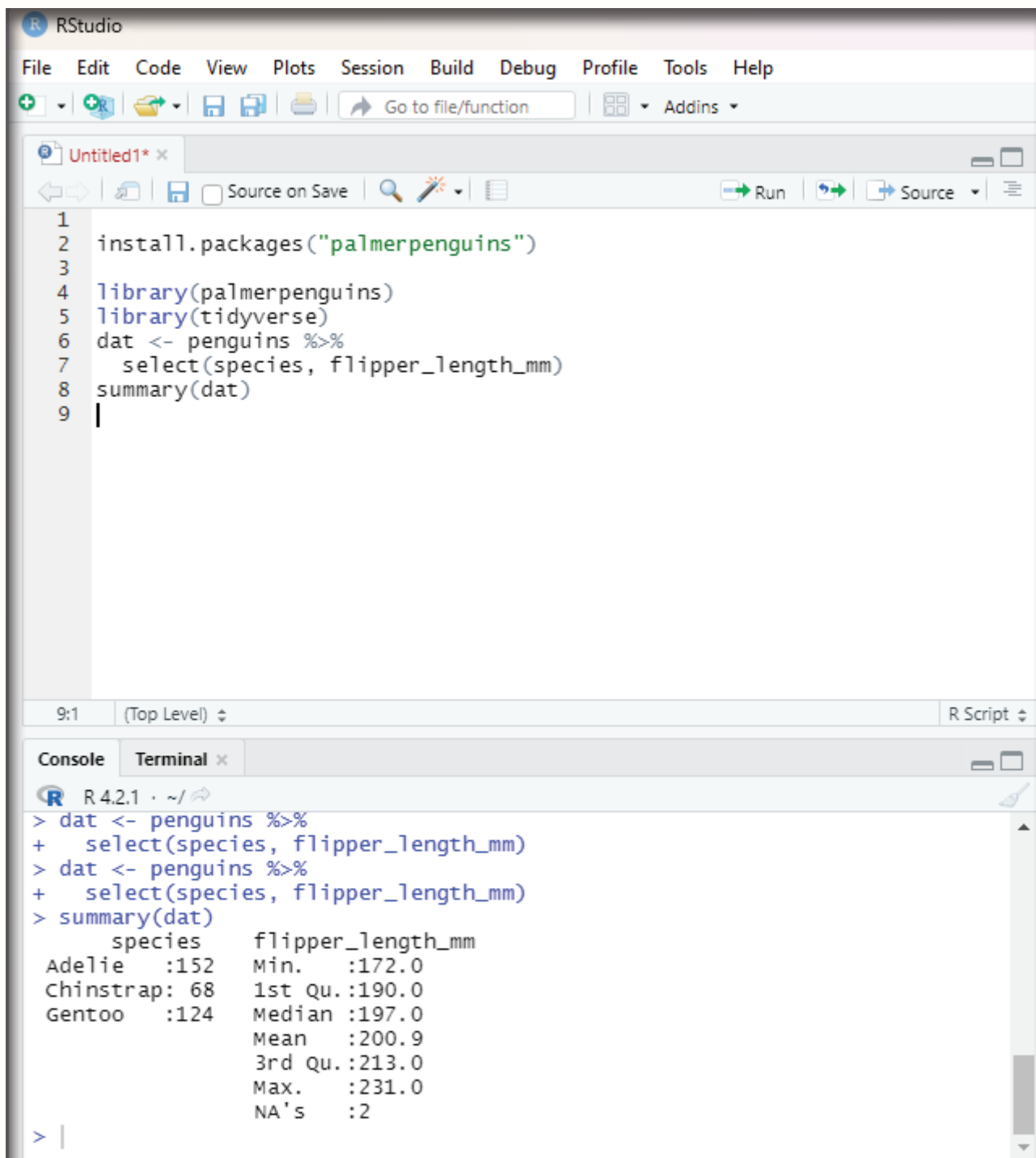
```
library(palmerpenguins)
```

In the next step to analyse the dataset package named tidyverse should be called into action. As described in previous chapter it should be installed first.

```
library(tidyverse)
```

In the example dataset penguins there are data for 344 penguins belonging to three different species. The dataset contains 8 variables, but the focus is only on the flipper length and the species. Only these two variables are taken up for comparison.

```
dat <- penguins %>%  
  select(species, flipper_length_mm)  
  summary(dat)
```



The image shows the RStudio interface. The top menu bar includes File, Edit, Code, View, Plots, Session, Build, Debug, Profile, Tools, and Help. Below the menu is a toolbar with icons for file operations and a search bar. The main editor window, titled 'Untitled1*', contains the following R code:

```
1  
2 install.packages("palmerpenguins")  
3  
4 library(palmerpenguins)  
5 library(tidyverse)  
6 dat <- penguins %>%  
7   select(species, flipper_length_mm)  
8 summary(dat)  
9 |
```

Below the editor is a console window showing the execution of the code. The prompt is 'R 4.2.1 · ~/'. The output of the `summary(dat)` command is displayed as follows:

```
> dat <- penguins %>%  
+   select(species, flipper_length_mm)  
> dat <- penguins %>%  
+   select(species, flipper_length_mm)  
> summary(dat)  
   species  flipper_length_mm  
Adelie   :152   Min.   :172.0  
Chinstrap: 68   1st Qu.:190.0  
Gentoo   :124   Median :197.0  
          Mean    :200.9  
          3rd Qu.:213.0  
          Max.    :231.0  
          NA's    :2
```

Image showing the response to command `summary(dat)` being displayed

Summary statistic reveals:

Flipper lengths varies from 172 to 231 mm, with a mean of 200.9 mm.

There are 152 Adelie penguins

68 Chinstrap penguins and

124 Gentoo penguins.

Visual representation of the summary:

```
# library ggplot should be called up
```

```
library(ggplot)
```

```
ggplot(dat) +
```

```
aes(x = species, y = flipper_length_mm, color = species) + geom_jitter() + theme(legend.position  
= "none")
```

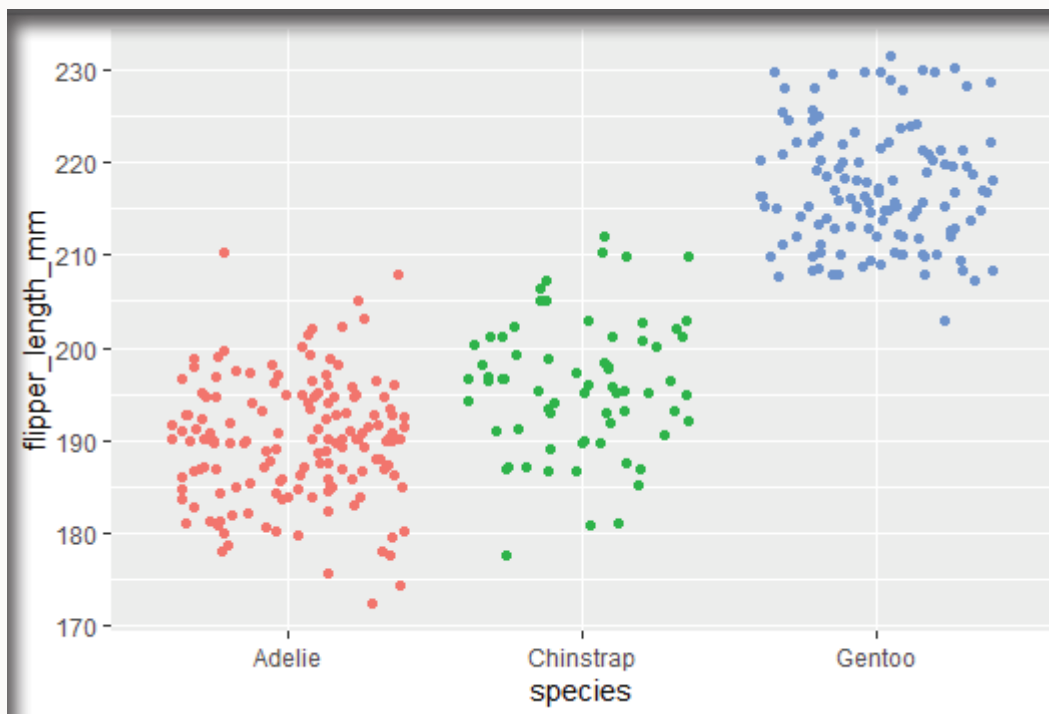


Image showing scatter plot displaying the summary

The aim of the analysis is to use ANOVA to answer the question “Is the length of the flippers different between the 3 species of penguins?”

Null hypothesis:

The three species are equal in terms of flipper length.

Alternate hypothesis:

At least one species is different from the other 2 in terms of flipper length.

In the database under discussion, the dependent variable is flipper_length_mm and the independent variable happens to be species. Species is a qualitative variable with 3 levels corresponding to the 3 species. Since there is a mix of two variables the basic assumption of ANOVA is met.

Independence of the observations is assumed as the data have been collected from a randomly selected portion of the population and measurements within and between the samples are not related.

```
res_aov <- aov(flipper_length_mm ~ species,  
              data = dat  
              )
```

Normality of data can be checked for visually:

```
par(mfrow = c(1, 2)) # combine plots  
  
# histogram  
hist(res_aov$residuals)  
  
# QQ-plot  
library(car)  
qqPlot(res_aov$residuals,  
       id = FALSE # id = FALSE to remove point identification  
       )
```

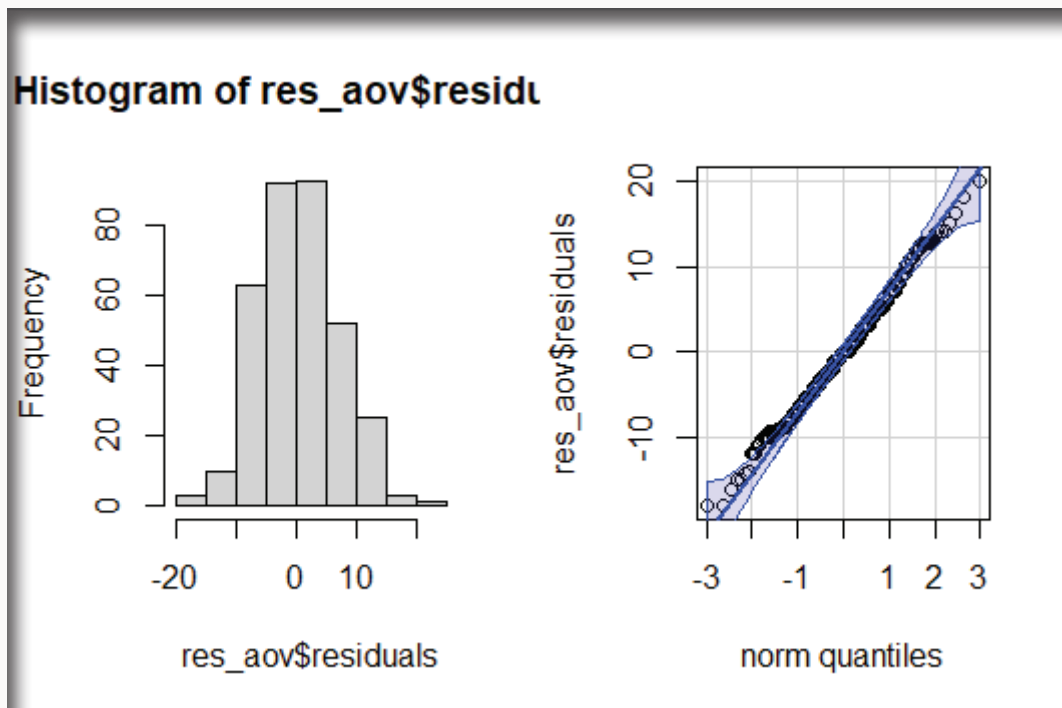


Image showing Graph generated that can be used to check the normality of the dataset

From the histogram and QQ-plot above, one can see that the normality assumption of the data seems to have been met. Histogram roughly forms the bell shaped curve.

Normality test:

This includes visual test that has been described above and statistical normality tests. Some researchers insist that normality should be tested both visually and statistically.

Anova tests are very robust to small deviations from normality. It can be quite conservative, in rejecting the null hypothesis. This is evident while testing large sample size.

Shapiro test can be used to ascertain the normality of the data. Shapiro function is usually written as `shapiro.test()`.

```
shapiro.test(res_ano$residuals)
```

If the p-value of Shapiro-Wick test on the residuals is larger than the usual significance level of $\alpha = 5\%$ the null hypothesis is not rejected.

Tests for equality of variances (homogeneity):

Assuming that residuals follow normal distributions, one should check whether the variances are equal across species or not. The result will help the user to decide whether to use ANOVA or Welch ANOVA. Visually this can be verified via a boxplot or dotplot or by a statistical test (Levene's test).

```
# Boxplot  
boxplot(flipper_length_mm ~ species,  
data = dat  
)
```

```
# Dotplot  
library("lattice")  
  
dotplot(flipper_length_mm ~ species,  
data = dat  
)
```

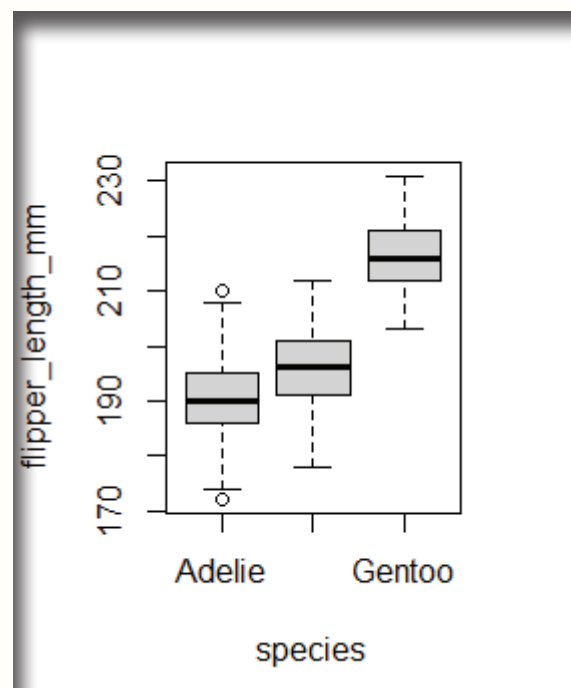


Image showing Boxplot generated

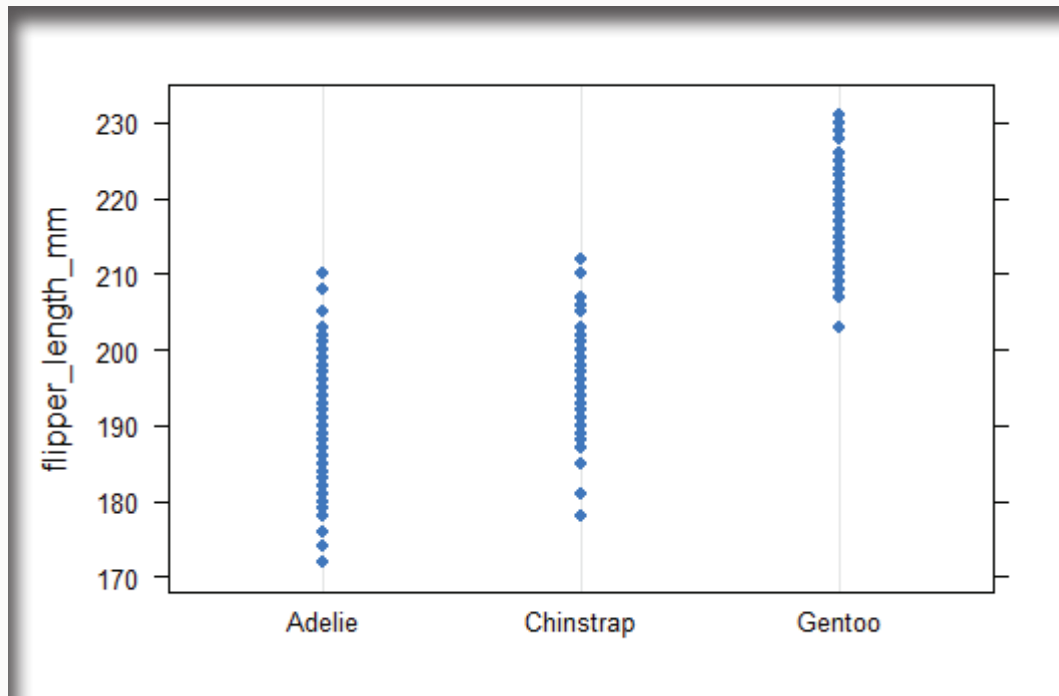
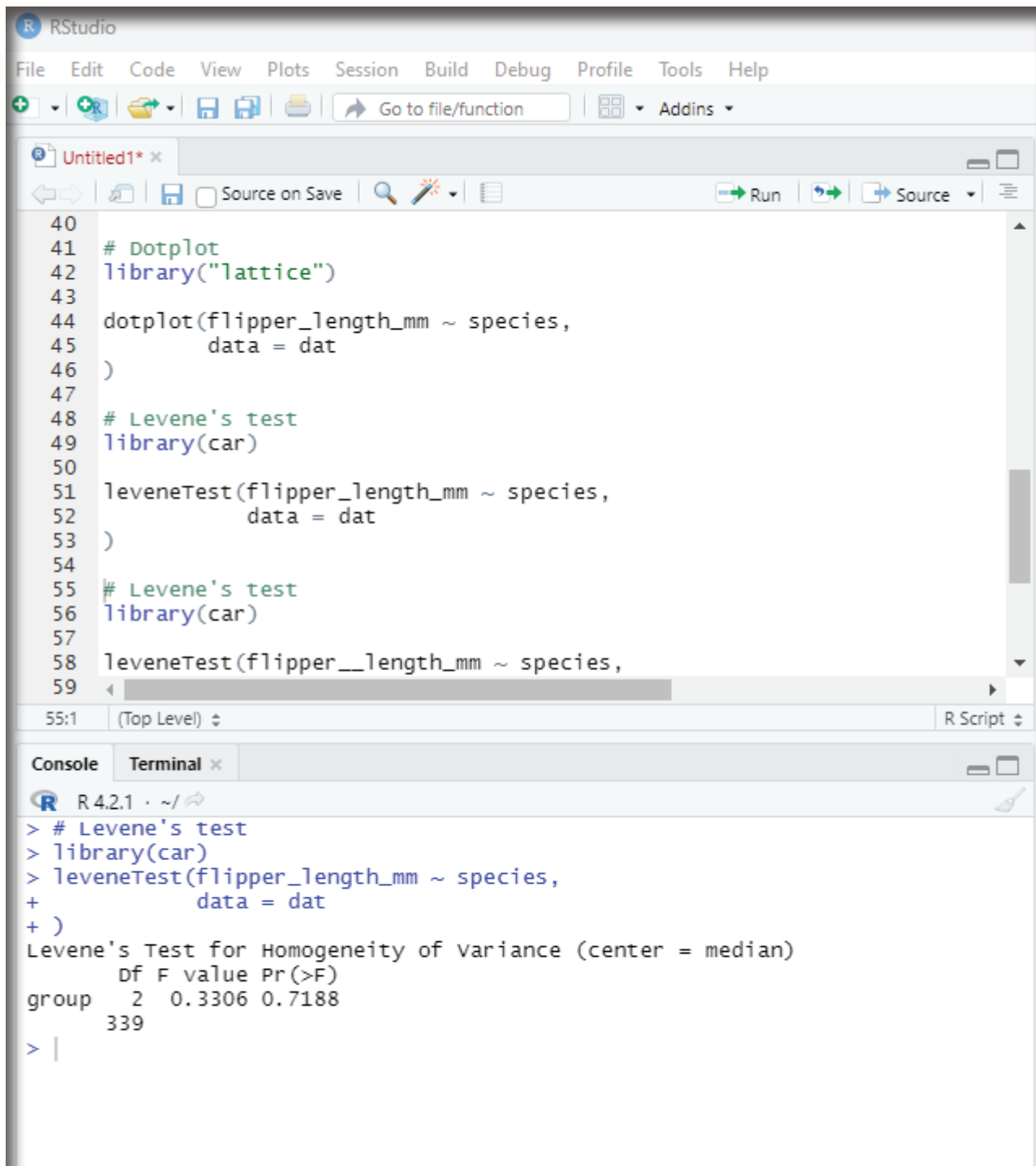


Image showing dotplot generated

In R Levene's test can be performed using `leveneTest()` function from the `{car}` package.

```
# Levene's test  
library(car)  
  
leveneTest(flipper_length_mm ~ species,  
data = dat  
)
```

The image is a screenshot of the RStudio interface. The top pane shows an R script with the following code:

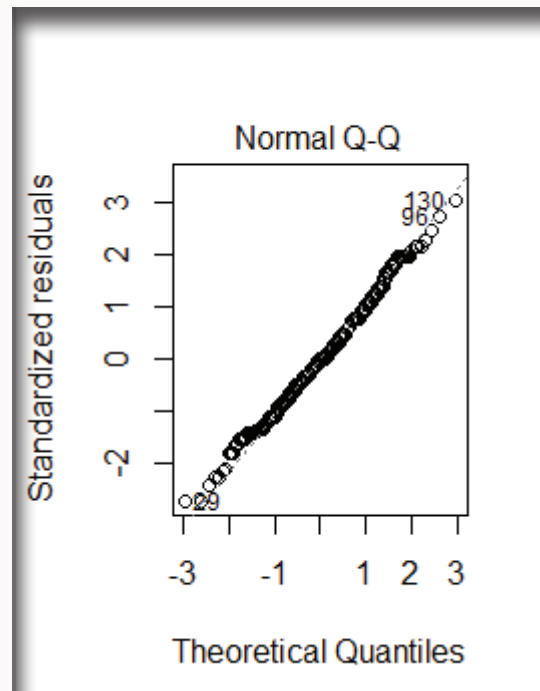
```
40  
41 # Dotplot  
42 library("lattice")  
43  
44 dotplot(flipper_length_mm ~ species,  
45         data = dat  
46 )  
47  
48 # Levene's test  
49 library(car)  
50  
51 leveneTest(flipper_length_mm ~ species,  
52            data = dat  
53 )  
54  
55 # Levene's test  
56 library(car)  
57  
58 leveneTest(flipper__length_mm ~ species,  
59 )
```

The bottom pane shows the console output for the Levene's test:

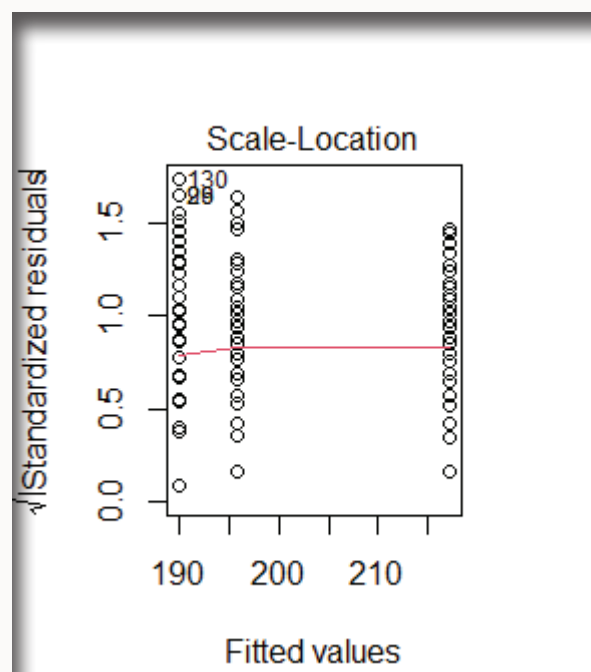
```
> # Levene's test  
> library(car)  
> leveneTest(flipper_length_mm ~ species,  
+            data = dat  
+ )  
Levene's Test for Homogeneity of Variance (center = median)  
      Df F value Pr(>F)  
group  2  0.3306 0.7188  
      339  
> |
```

Image showing Levene's test

Levene's test reveals that the p-value is larger than the significance level of 0.05 the null hypothesis is not rejected. The null hypothesis states that the variances are equal between species (p-value = 0.719).



Graph showing Theoretical Quantiles



Graph showing fitted values

plot() function is another method that can be used to test normality and homogeneity of dataset.

```
par(mfrow = c(1, 2)) # combine plots
```

```
# 1. Homogeneity of variances
```

```
plot(res_aov, which = 3)
```

```
# 2. Normality
```

```
plot(res_aov, which = 2)
```

Outliers:

There are several techniques available to detect outliers. Boxplot is an useful visual approach for the same.

```
boxplot(flipper_length_mm ~ species,  
data = dat  
)
```

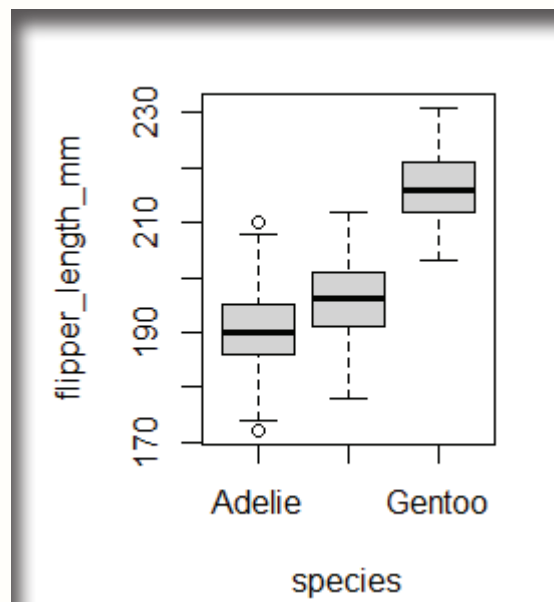


Image showing boxplot

ggplot package can also be used for this purpose.

library(ggplot2)

*ggplot(dat) +
aes(x = species, y = flipper_length_mm) +
geom_boxplot()*

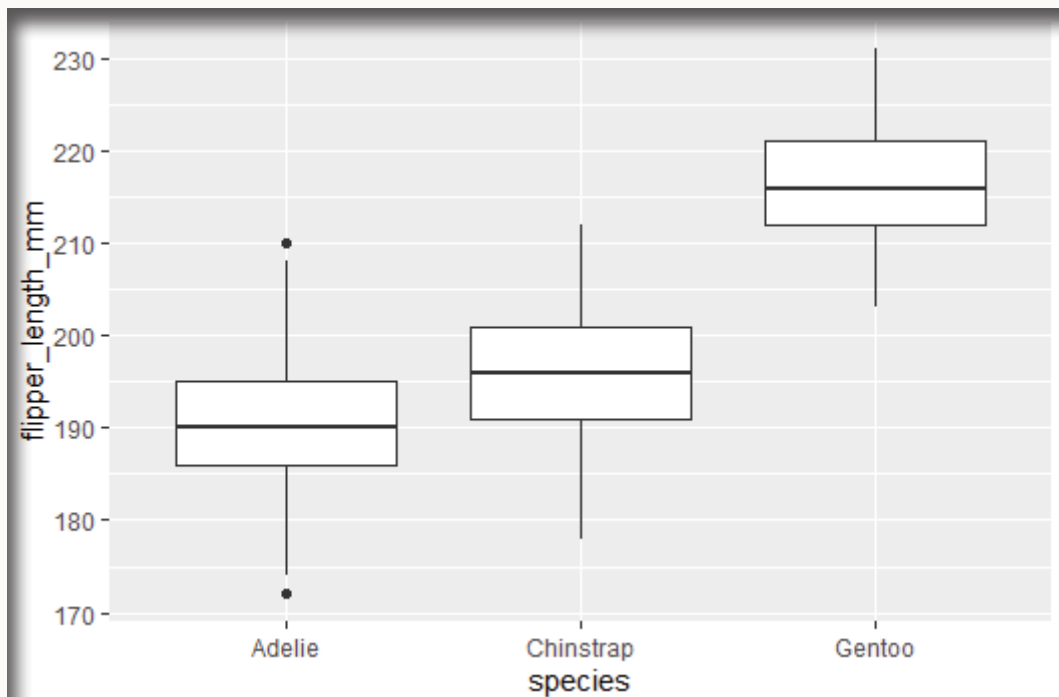


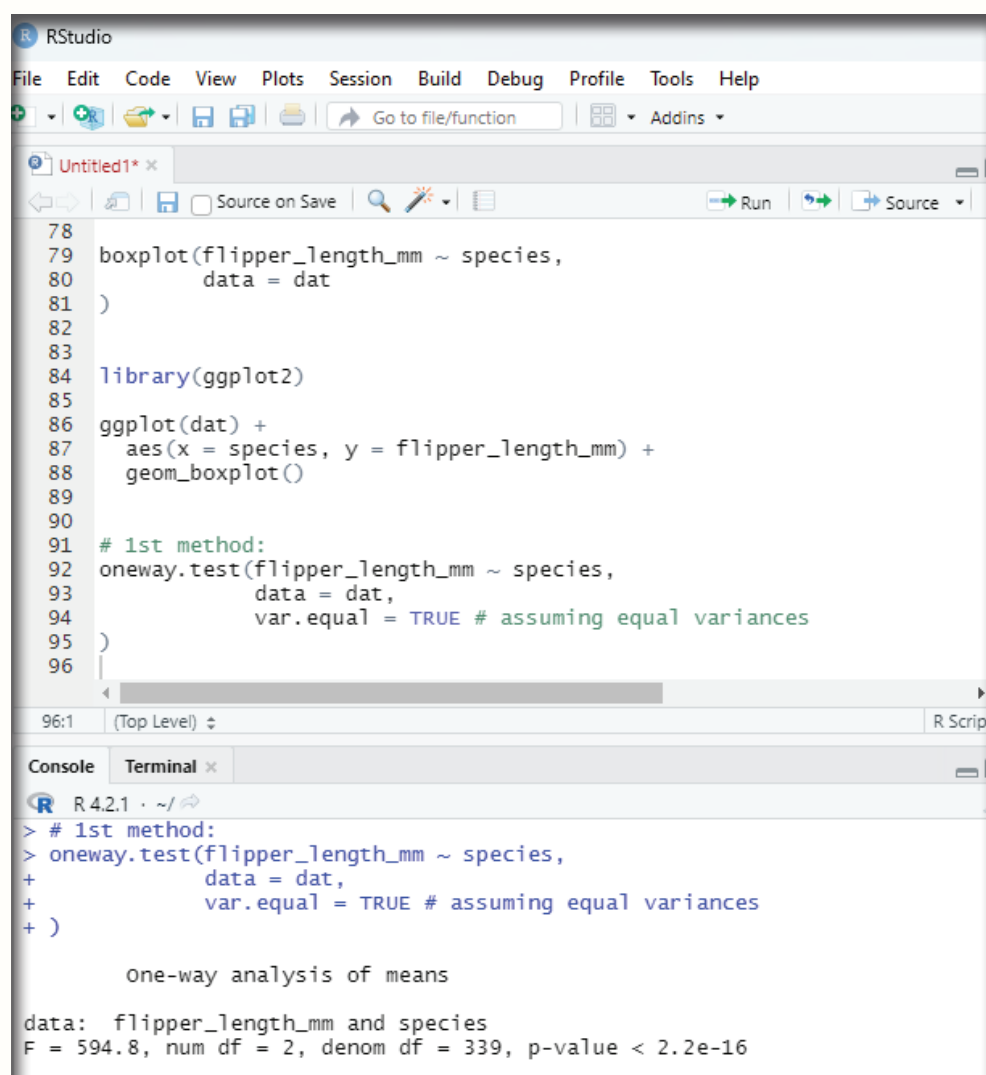
Image showing ggplot constructed

Using ANOVA to answer the question “Is the length of the flippers different between the 3 species of penguins?”

`oneway.test()` function can be used.

1st method:

```
oneway.test(flipper_length_mm ~ species,  
            data = dat,  
            var.equal = TRUE # assuming equal variances  
            )
```



The screenshot shows the RStudio interface. The script editor contains the following code:

```
78 boxplot(flipper_length_mm ~ species,  
79         data = dat  
80 )  
81  
82  
83  
84 library(ggplot2)  
85  
86 ggplot(dat) +  
87   aes(x = species, y = flipper_length_mm) +  
88   geom_boxplot()  
89  
90  
91 # 1st method:  
92 oneway.test(flipper_length_mm ~ species,  
93             data = dat,  
94             var.equal = TRUE # assuming equal variances  
95 )  
96
```

The console output shows the execution of the `oneway.test` function:

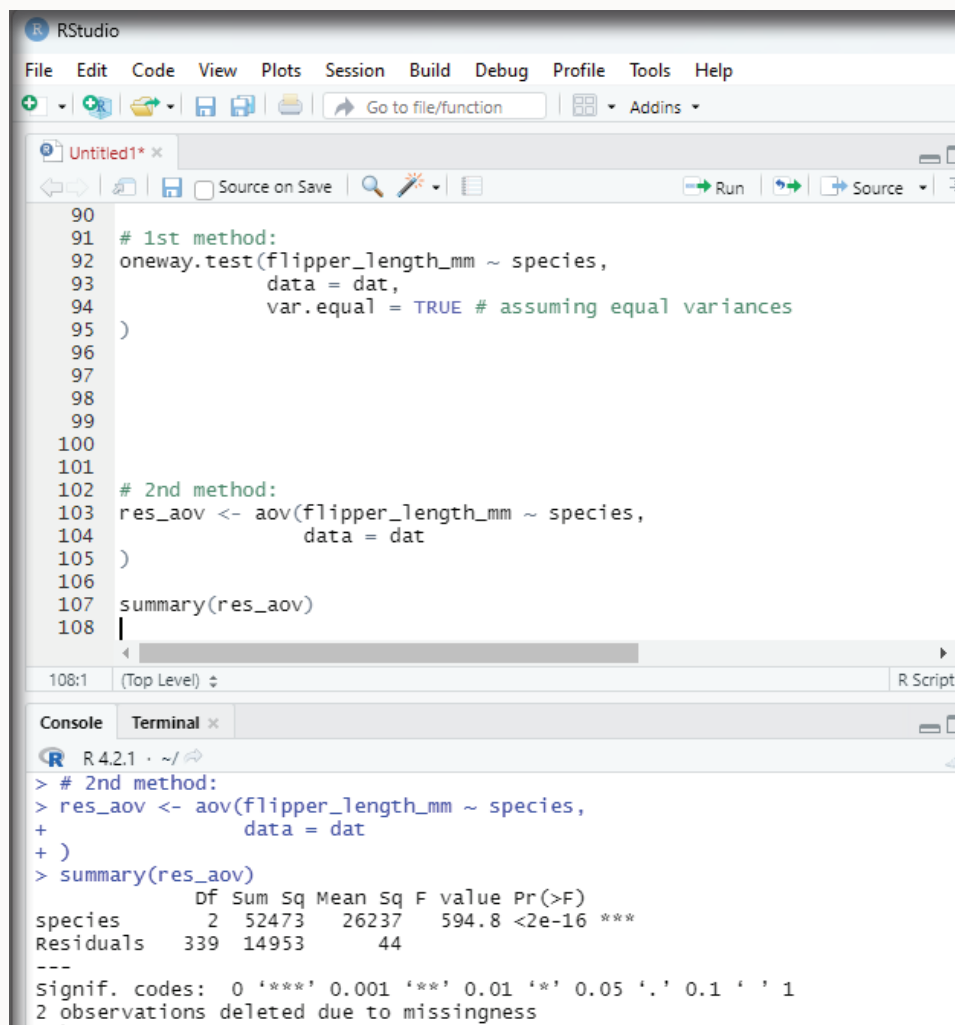
```
> # 1st method:  
> oneway.test(flipper_length_mm ~ species,  
+             data = dat,  
+             var.equal = TRUE # assuming equal variances  
+ )  
  
One-way analysis of means  
  
data: flipper_length_mm and species  
F = 594.8, num df = 2, denom df = 339, p-value < 2.2e-16
```

Image showing first method being performed

2nd method:

This method uses `summary()` and `aov()` functions.

```
# 2nd method:  
res_aov <- aov(flipper_length_mm ~ species,  
              data = dat  
              )  
  
summary(res_aov)
```



The screenshot shows the RStudio interface with a script editor and a console. The script editor contains the following code:

```
90  
91 # 1st method:  
92 oneway.test(flipper_length_mm ~ species,  
93             data = dat,  
94             var.equal = TRUE # assuming equal variances  
95 )  
96  
97  
98  
99  
100  
101  
102 # 2nd method:  
103 res_aov <- aov(flipper_length_mm ~ species,  
104               data = dat  
105               )  
106  
107 summary(res_aov)  
108 |
```

The console shows the output of the `summary(res_aov)` command:

```
> # 2nd method:  
> res_aov <- aov(flipper_length_mm ~ species,  
+               data = dat  
+               )  
> summary(res_aov)
```

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
species	2	52473	26237	594.8	<2e-16 ***
Residuals	339	14953	44		

Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
2 observations deleted due to missingness

Image showing second method performed

As can be seen from the two outputs above, the test statistic ($F =$ in the first method and F value in the second one) and the p-value (p-value in the first method and $\text{Pr}(>F)$ in the second one) are exactly the same for both methods, which means that in case of equal variances, results and conclusions will be unchanged.

Interpretation of ANOVA results:

If the p-value is smaller than 0.05 the null hypothesis which assumes that all means are equal stands rejected. It can hence be concluded that at least one species is different than the others in terms of flippers length.

If the p-value is greater than 0.05 then the null hypothesis is not rejected. It can now be assumed that all groups are equal.

Post-hoc tests in R:

These are a battery of tests performed to deal with the problem when null hypothesis has been rejected after performing ANOVA. As the number of groups increase, the number of comparisons also increases and the probability of having a significant result simply due to chance keeps increasing. Post-hoc tests take into account this scenario by adjusting the alpha value in some way, so that the probability of observing at least one significant result due to chance remains below the selected or desired significance level.

Common Post-hoc tests used include:

1. Tukey HSD - This test is used to compare all groups to each other (it delivers comparative values of all possible 2 groups).
2. Dunnett test - This test is used to make comparisons with a reference group. The reference group can also be called as a control group.
3. Bonferroni correction - This can be used if there is a set of planned comparisons to do.

Tukey HSD test:

```
library(multcomp)
```

```
# Tukey HSD test:
```

```
post_test <- glht(res_aov, linfct = mcp(species = "Tukey"))
```

```
summary(post_test)
```

Library multcomp needs to be installed.

Example code for running Dunnett's test:

```
library(multcomp)

# Dunnett's test:
post_test <- glht(res_aov,
  linfct = mcp(species = "Dunnett")
)

summary(post_test)
```

Descriptive Statistics

Descriptive statistics aims at summarizing, describing and presenting a series of values or a dataset. This is often the first step and a very important one in any statistical analysis.

1. It allows the user to check the quality of the data.
2. It helps the user to have a clear understanding of the dataset.
3. If performed well it serves as a starting point for further data analysis.

Measures used to summarize the dataset are of two types:

1. Location measures
2. Dispersion measures

Location measures give an understanding about the central tendency of the data, while the dispersion measures give an understanding about the spread of the data.

Dataset used in this chapter is iris which is inbuilt and available within R environment. This dataset can be loaded by running iris:

Code:

```
# Loading iris dataset
```

```
dat <- iris
```

```
# Getting a preview of the dataset and its structure.
```

```
# The command below reveals the first 6 observations.
```

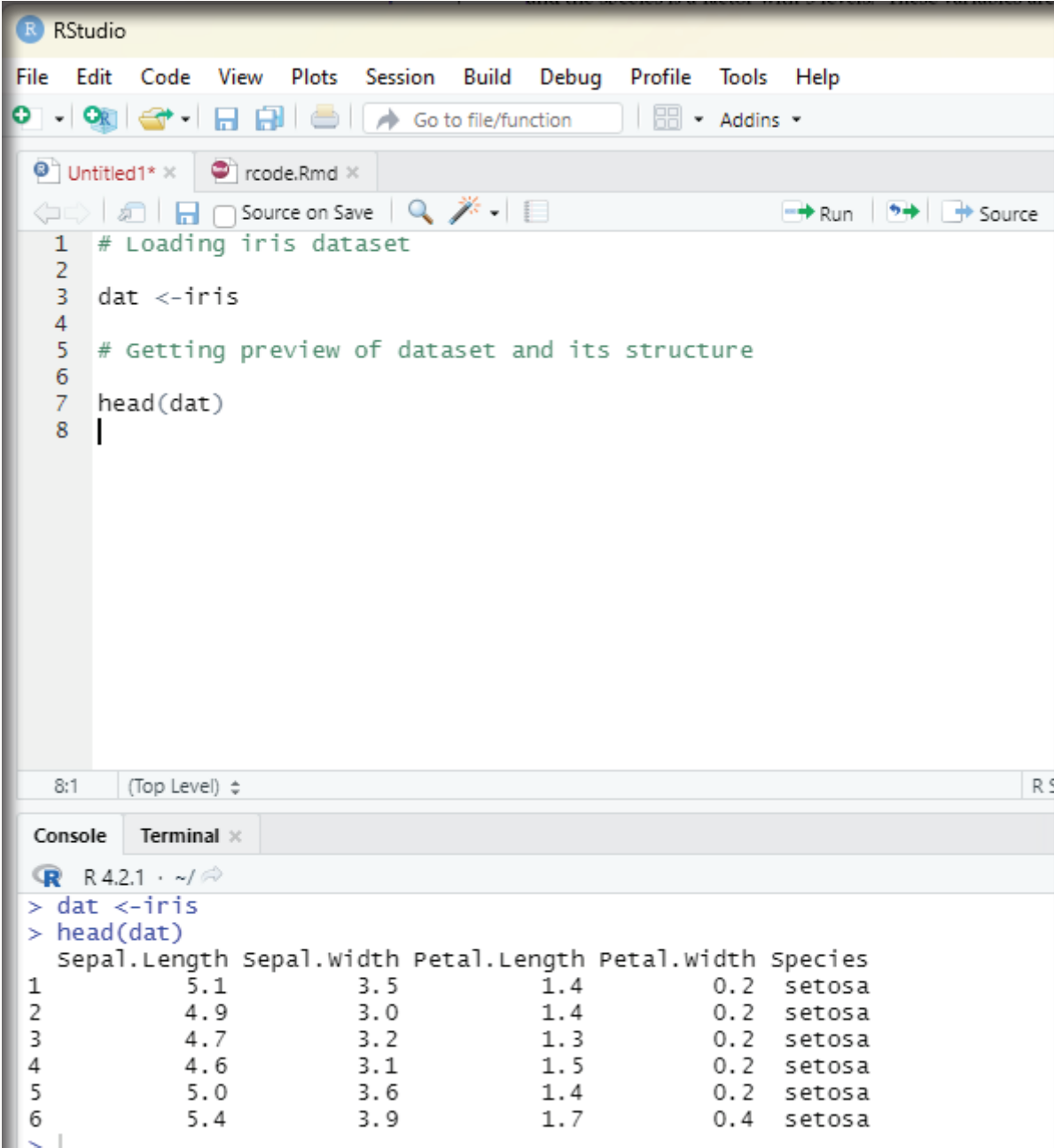
```
head(dat)
```

```
# Next is getting the structure of the dataset
```

```
str(dat)
```

The dataset iris contains 150 observations and 5 variables, representing the length and width of the sepal and petal and the species of 150 flowers. The length and the width of the sepal and petal are numeric variables

and the species is a factor with 3 levels. These variables are indicated by (num) and (Factor w/ 3 levels) after the name of the variables.



The screenshot shows the RStudio interface. The source editor contains the following R code:

```
1 # Loading iris dataset
2
3 dat <- iris
4
5 # Getting preview of dataset and its structure
6
7 head(dat)
8 |
```

The console output shows the result of the code execution:

```
> dat <- iris
> head(dat)
  Sepal.Length Sepal.width Petal.Length Petal.width species
1          5.1         3.5         1.4         0.2   setosa
2          4.9         3.0         1.4         0.2   setosa
3          4.7         3.2         1.3         0.2   setosa
4          4.6         3.1         1.5         0.2   setosa
5          5.0         3.6         1.4         0.2   setosa
6          5.4         3.9         1.7         0.4   setosa
```

Image showing iris dataset loaded

Minimum and maximum:

These values can be ascertained using min() and max() functions:

```
min(dat$Sepal.Length)
```

```
max(dat$Sepal.Length)
```

Alternatively range() function can also be used:

```
rng <- range(dat$Sepal.Length)  
rng
```

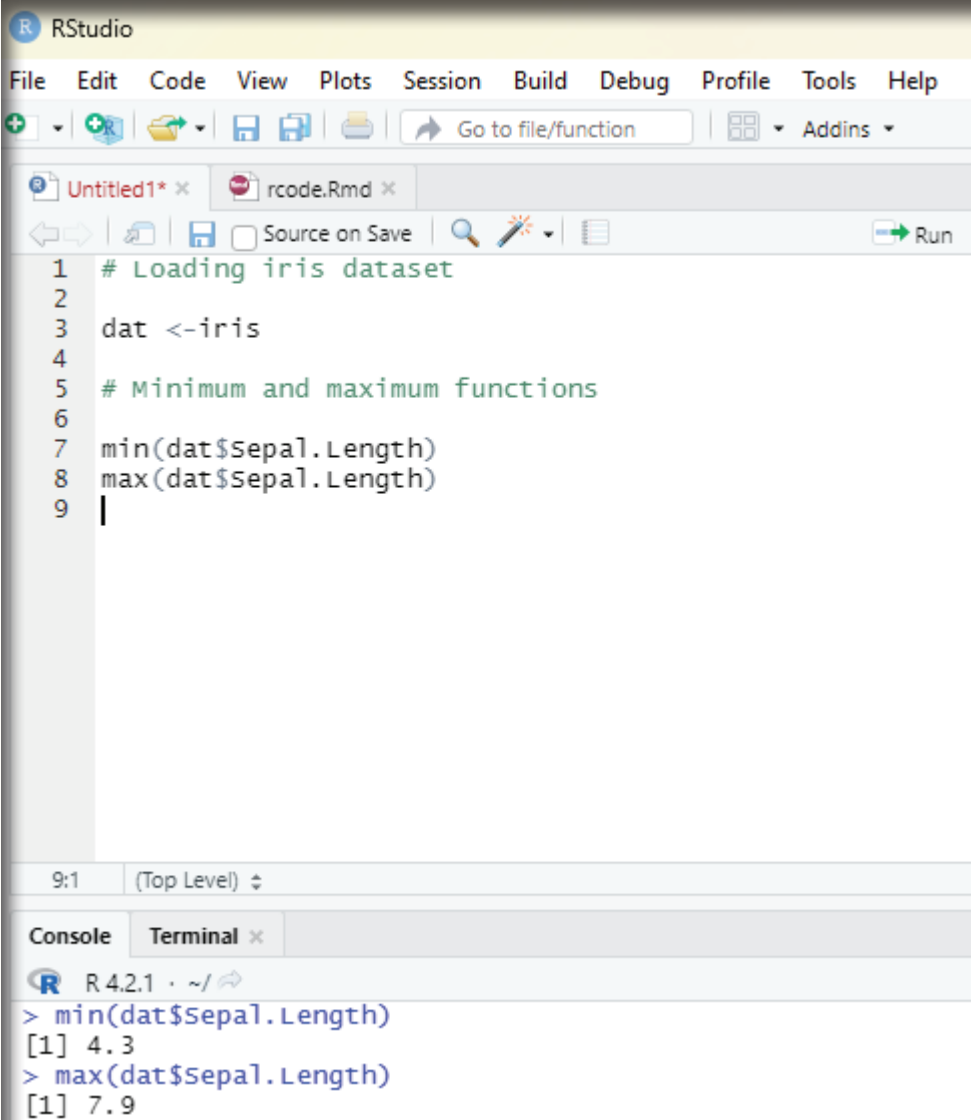
The function range gives the minimum and maximum directly in that order.

Using range() function one can access the minimum with the following code:

```
rng[1]
```

Using range() function one can access the maximum with the following code:

```
rng[2]
```



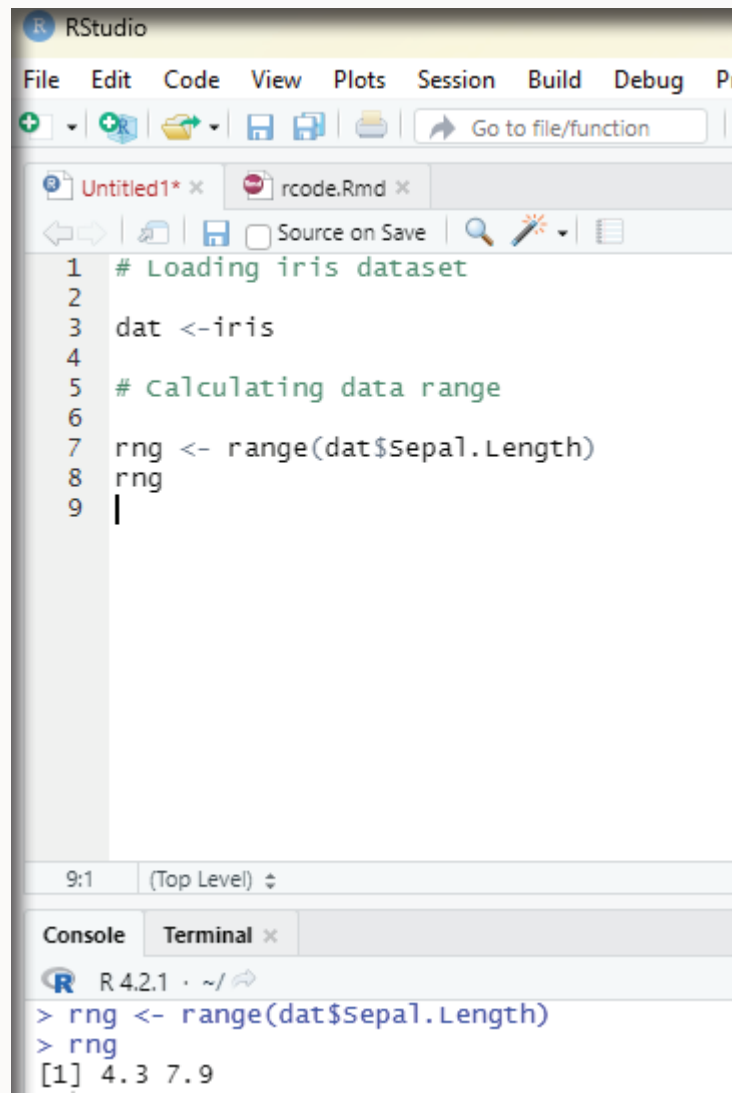
The screenshot shows the RStudio interface. The source editor contains the following R code:

```
1 # Loading iris dataset  
2  
3 dat <- iris  
4  
5 # Minimum and maximum functions  
6  
7 min(dat$Sepal.Length)  
8 max(dat$Sepal.Length)  
9 |
```

The console at the bottom shows the output of the last two commands:

```
> min(dat$Sepal.Length)  
[1] 4.3  
> max(dat$Sepal.Length)  
[1] 7.9
```

Image showing calculation of minimum and maximum values



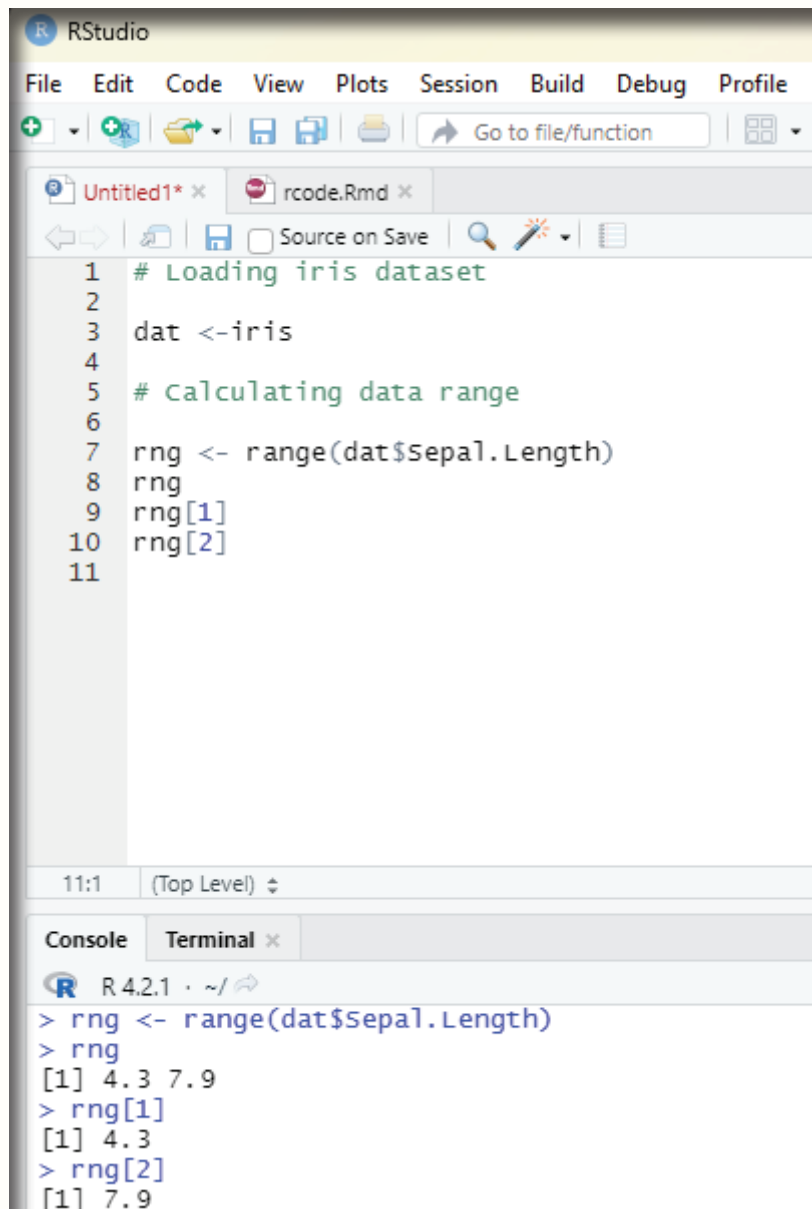
The image shows the RStudio interface. The top menu bar includes File, Edit, Code, View, Plots, Session, Build, Debug, and Profiler. Below the menu is a toolbar with icons for file operations and a search bar labeled 'Go to file/function'. The editor pane shows a script with the following code:

```
1 # Loading iris dataset
2
3 dat <- iris
4
5 # calculating data range
6
7 rng <- range(dat$Sepal.Length)
8 rng
9 |
```

Below the editor is a status bar showing '9:1' and '(Top Level)'. The console pane at the bottom shows the output of the code:

```
R 4.2.1 · ~/
> rng <- range(dat$Sepal.Length)
> rng
[1] 4.3 7.9
```

Image showing calculation of range



The image shows the RStudio interface. The top menu bar includes File, Edit, Code, View, Plots, Session, Build, Debug, and Profile. Below the menu bar is a toolbar with icons for file operations and a search bar labeled 'Go to file/function'. The main editor window displays a script with the following R code:

```
1 # Loading iris dataset
2
3 dat <- iris
4
5 # calculating data range
6
7 rng <- range(dat$Sepal.Length)
8 rng
9 rng[1]
10 rng[2]
11
```

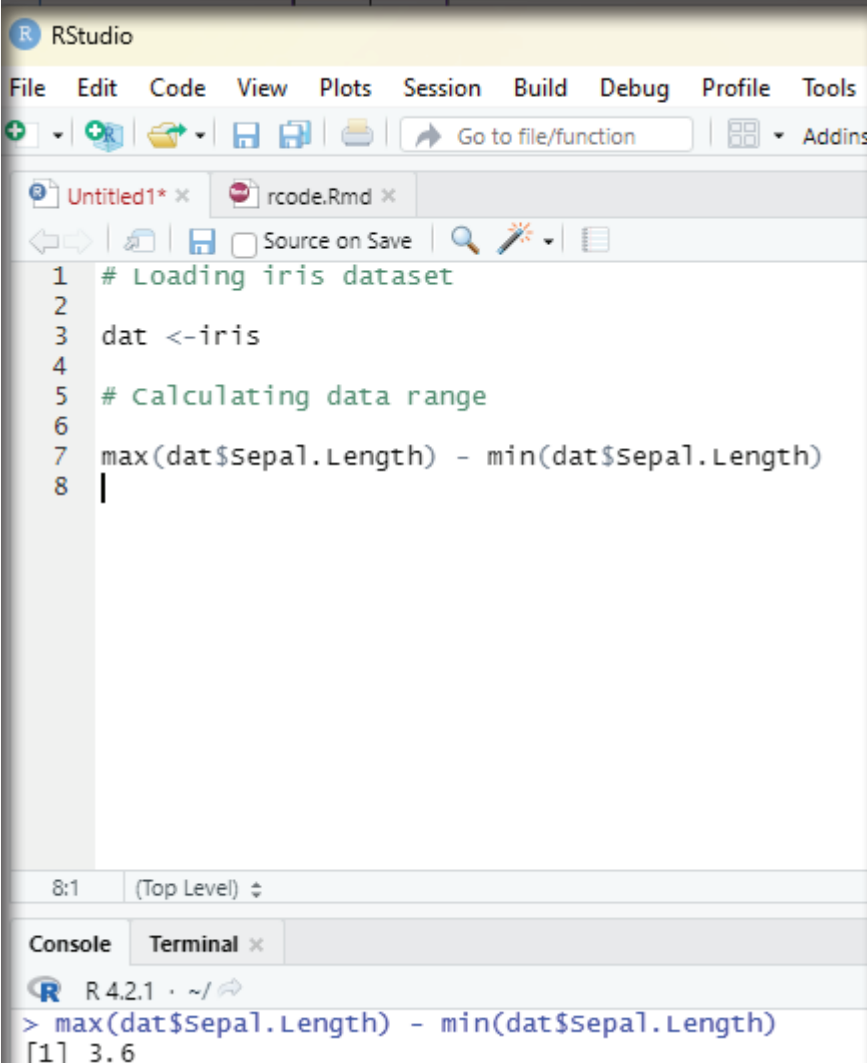
Below the editor window is a status bar showing '11:1' and '(Top Level)'. At the bottom, there are two tabs: 'Console' and 'Terminal'. The 'Console' tab is active, showing the following output:

```
R 4.2.1 · ~/
> rng <- range(dat$Sepal.Length)
> rng
[1] 4.3 7.9
> rng[1]
[1] 4.3
> rng[2]
[1] 7.9
```

Image showing rng[1] and rng[2] command result

Range can also be computed by subtracting the minimum from the maximum:

```
max(dat$Sepal.Length) - min(dat$Sepal.Length)
```



The screenshot shows the RStudio interface. The source editor on the left contains the following R code:

```
1 # Loading iris dataset
2
3 dat <- iris
4
5 # calculating data range
6
7 max(dat$Sepal.Length) - min(dat$Sepal.Length)
8 |
```

The console at the bottom shows the execution of the command on line 7, resulting in the output:

```
> max(dat$Sepal.Length) - min(dat$Sepal.Length)
[1] 3.6
```

Image showing calculation of range between maximum and minimum values

Mean:

Mean value of a dataset can be computed using mean() function.

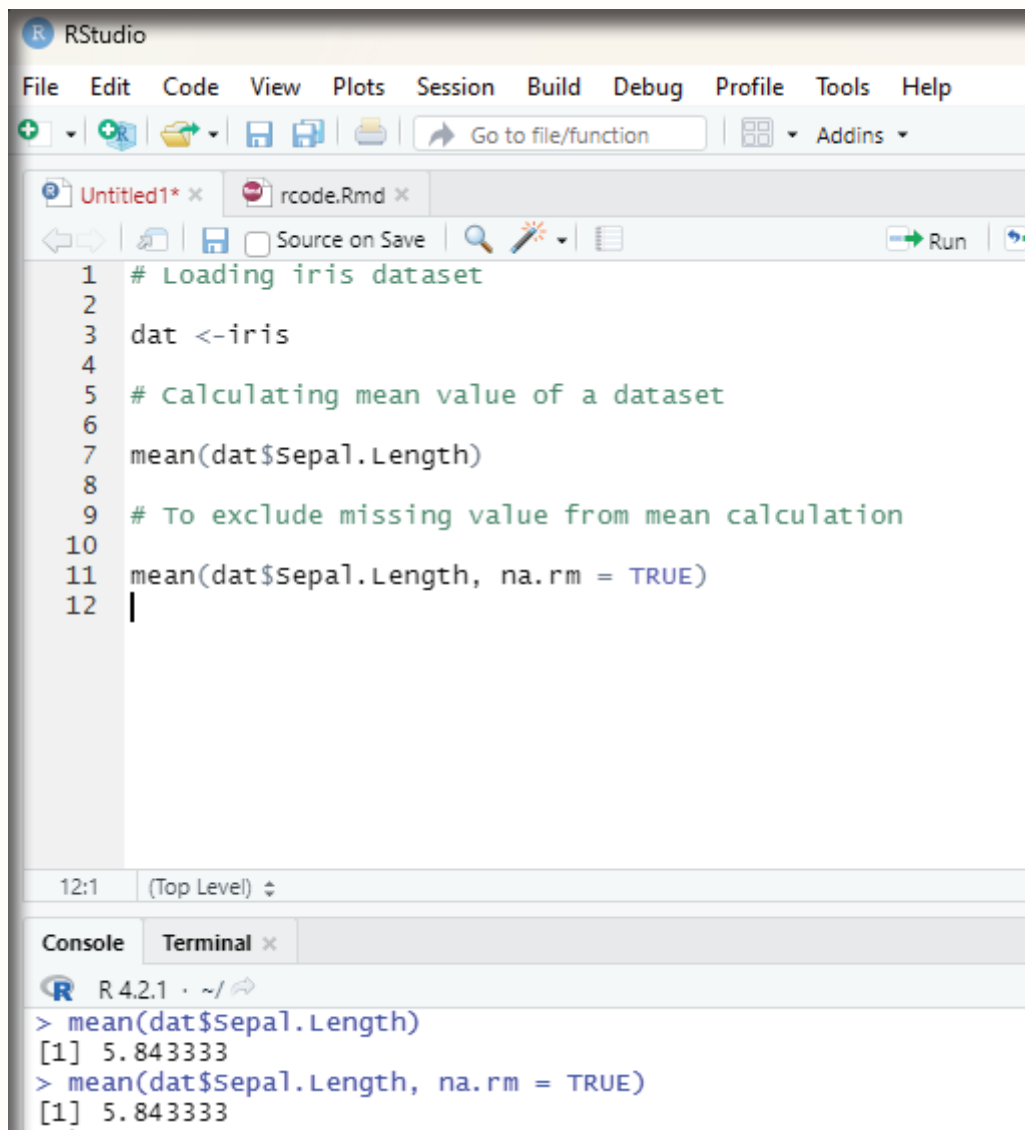
code:

```
mean(dat$Sepal.Length)
```

If there is even one missing value in the dataset then it needs to be excluded while calculating the mean. The following code should be used:

```
mean(dat$Sepal.Length, na.rm = TRUE)
```

In order to get a truncated mean value then the following code should be used:



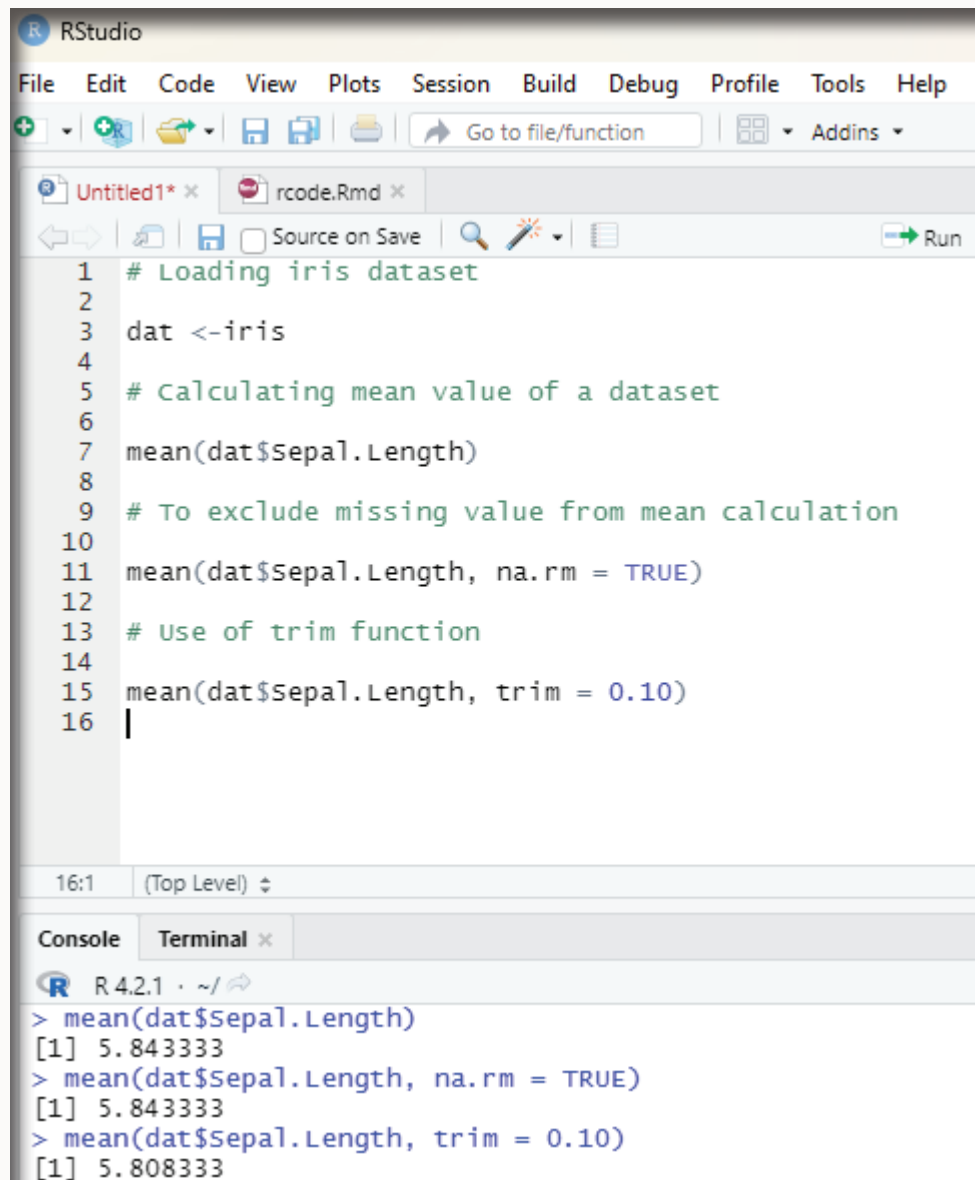
```
RStudio
File Edit Code View Plots Session Build Debug Profile Tools Help
+ [Go to file/function] Addins
Untitled1* x rcode.Rmd x
Source on Save Run
1 # Loading iris dataset
2
3 dat <- iris
4
5 # calculating mean value of a dataset
6
7 mean(dat$Sepal.Length)
8
9 # To exclude missing value from mean calculation
10
11 mean(dat$Sepal.Length, na.rm = TRUE)
12 |

12:1 (Top Level)
Console Terminal x
R 4.2.1 ~
> mean(dat$Sepal.Length)
[1] 5.843333
> mean(dat$Sepal.Length, na.rm = TRUE)
[1] 5.843333
```

Image showing calculation of mean

mean(dat\$Sepal.Length, trim = 0.10)

The trim argument can be tweaked as per needs.



```
1 # Loading iris dataset
2
3 dat <- iris
4
5 # calculating mean value of a dataset
6
7 mean(dat$Sepal.Length)
8
9 # To exclude missing value from mean calculation
10
11 mean(dat$Sepal.Length, na.rm = TRUE)
12
13 # Use of trim function
14
15 mean(dat$Sepal.Length, trim = 0.10)
16 |
```

16:1 (Top Level) ⚡

Console Terminal x

R 4.2.1 · ~/

```
> mean(dat$Sepal.Length)
[1] 5.843333
> mean(dat$Sepal.Length, na.rm = TRUE)
[1] 5.843333
> mean(dat$Sepal.Length, trim = 0.10)
[1] 5.808333
```

Image showing execution of trim function

Median:

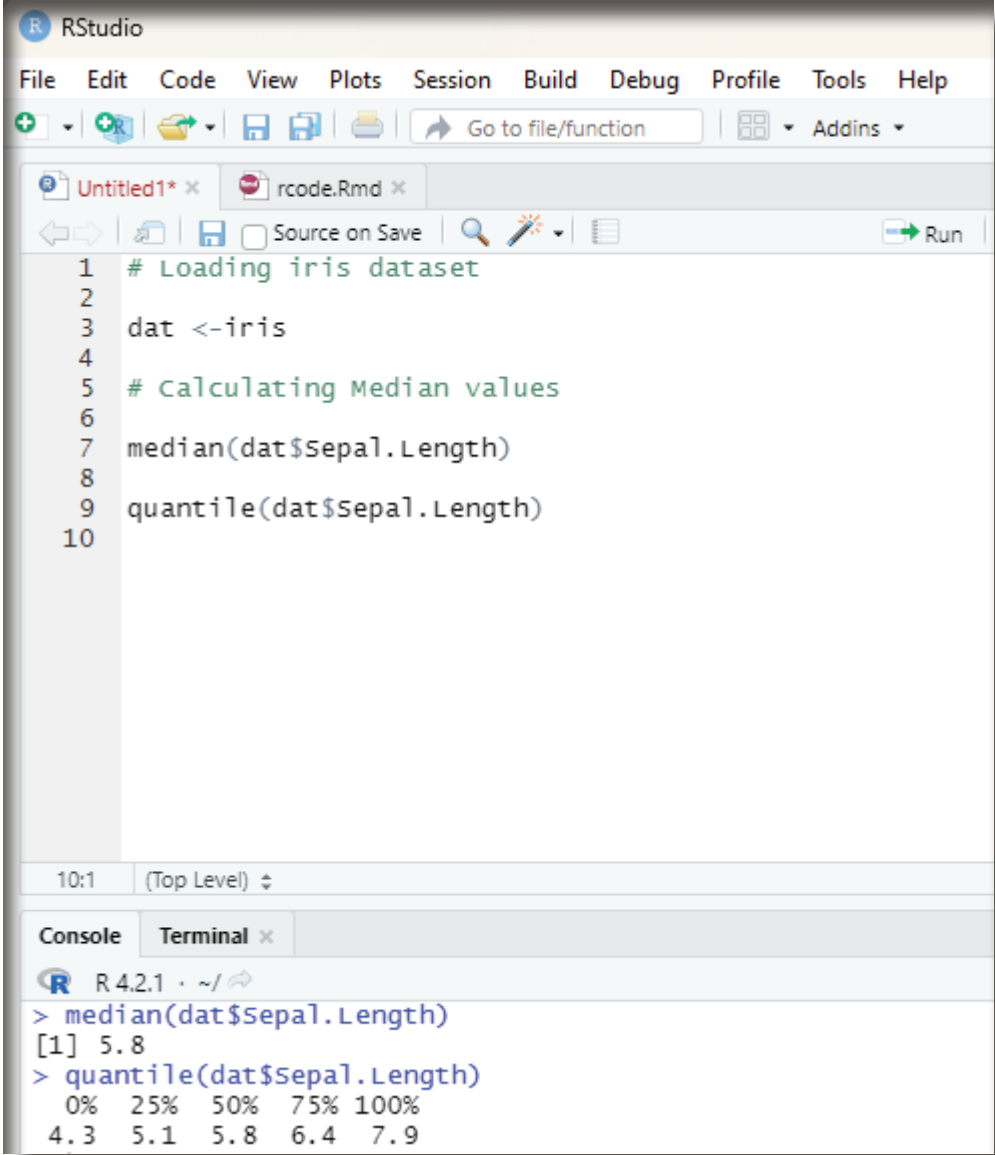
Median value of a dataset can be arrived at by using median() function.

```
median(dat$Sepal.Length)
```

Median value can also be arrived at by using quantile() function.

```
quantile(dat$Sepal.Length, 0.5)
```

Calculating First and third quartile:



The screenshot shows the RStudio interface with a script editor and a console. The script editor contains the following R code:

```
1 # Loading iris dataset
2
3 dat <- iris
4
5 # calculating median values
6
7 median(dat$Sepal.Length)
8
9 quantile(dat$Sepal.Length)
10
```

The console shows the output of the executed code:

```
> median(dat$Sepal.Length)
[1] 5.8
> quantile(dat$Sepal.Length)
 0%  25%  50%  75% 100%
4.3  5.1  5.8  6.4  7.9
```

Median and quantile calculations

This can be calculated using quantile() function and setting the second argument to 0.25 or 0.75.

```
# First quartile
```

```
quantile(dat$Sepal.Length, 0.25)
```

```
# Third quartile
```

```
quantile(dat$Sepal.Length, 0.75)
```

```
Code to generate 98th percentile:
```

```
quantile(dat$Sepal.Length, 0.98)
```

Interquartile range:

The interquartile range (i.e., the difference between the first and third quartile) can be computed with IQR() function.

```
IQR(dat$Sepal.Length)
```

Alternatively interquartile range can be calculated by using quantile() function also.

```
quantile(dat$Sepal.Length, 0.75) - quantile(dat$Sepal.Length, 0.25)
```

Standard deviation and variance:

These values can be computed by using sd() and var() functions.

```
# Calculation of standard deviation
```

```
sd(dat$Sepal.Length)
```

```
# Calculation of variance
```

```
var(dat$Sepal.Length)
```

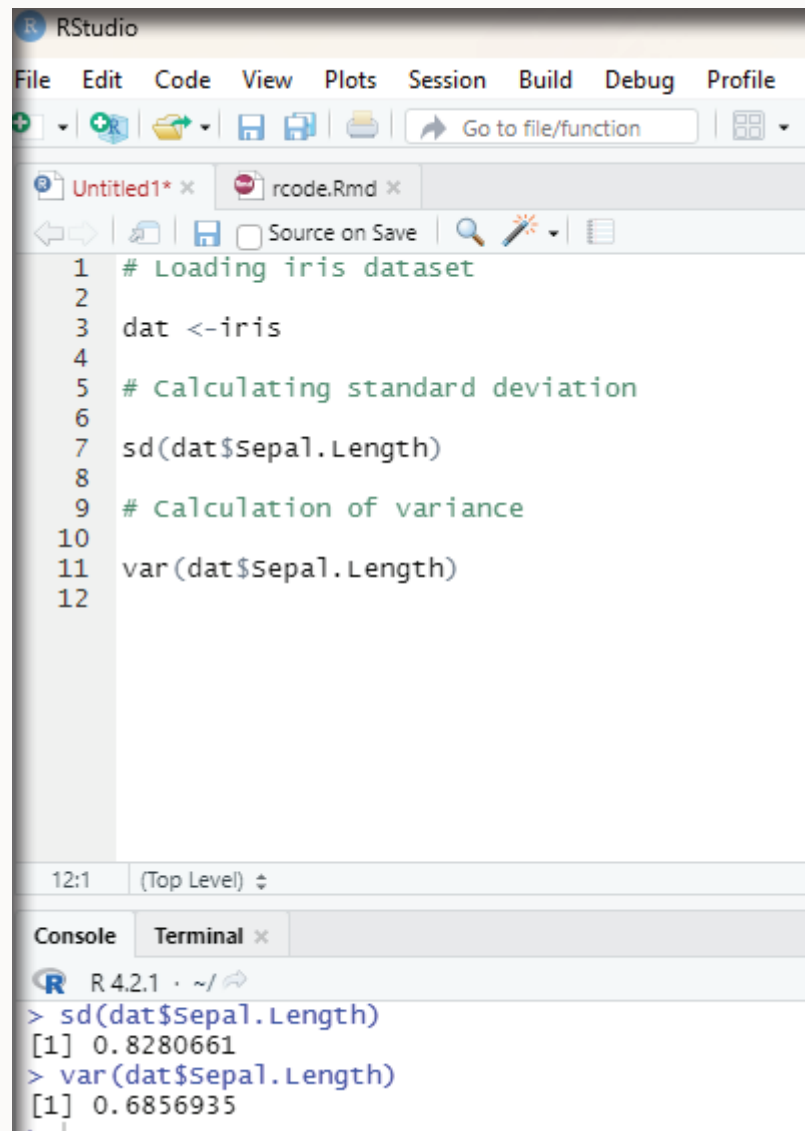
The image shows the RStudio interface with a script editor and a console. The script editor contains the following R code:

```
1 # Loading iris dataset
2
3 dat <-iris
4
5 # calculating first quartile
6 quantile(dat$Sepal.Length, 0.25)
7 # calculating third quartile
8 quantile(dat$Sepal.Length, 0.75)
9 # Code to generate 98th percentile
10 quantile(dat$Sepal.Length, 0.98)
11
```

The console shows the output of the code:

```
R 4.2.1 ~/> # calculating first quartile
> quantile(dat$Sepal.Length, 0.25)
25%
5.1
> # calculating third quartile
> quantile(dat$Sepal.Length, 0.75)
75%
6.4
> # Code to generate 98th percentile
> quantile(dat$Sepal.Length, 0.98)
98%
7.7
```

Image showing quartiles calculations



The image shows the RStudio interface. The top menu bar includes File, Edit, Code, View, Plots, Session, Build, Debug, Profile, and Tools. Below the menu is a toolbar with icons for file operations and a search bar labeled 'Go to file/function'. The editor pane shows a script with the following code:

```
1 # Loading iris dataset
2
3 dat <-iris
4
5 # calculating standard deviation
6
7 sd(dat$sepal.Length)
8
9 # calculation of variance
10
11 var(dat$sepal.Length)
12
```

The status bar at the bottom indicates the cursor is at line 12, column 1, at the top level. Below the editor is a console pane with the following output:

```
R 4.2.1 · ~/
> sd(dat$sepal.Length)
[1] 0.8280661
> var(dat$sepal.Length)
[1] 0.6856935
```

Image showing calculation of standard deviation and variance values of a dataset

To compute the standard deviation (or variance) of multiple variables at the same time, one can use lapply() function with appropriate statistics as second argument.

```
lapply(dat[, 1:4], sd)
```

Summary:

The user can compute the minimum, quartile, median, mean, and the maximum for all numerical variables of dataset using summary() function.

```
summary(dat)
```

If the user needs descriptive statistics by the group then by() function can be used.

```
by(dat, dat$Species, summary)
```

Coefficient of variation:

For this purpose the package pastecs needs to be installed and loaded.

```
install.packages("pastecs")
```

```
library(pastecs)
```

```
stat.desc(dat)
```

Manual calculation of coefficient of variation:

```
sd(dat$Sepal.Length) / mean(dat$Sepal.Length)
```

Mode:

R does not contain a function to find the mode of a variable. But, they can be found using functions table() and sort().

```
# Number of occurrences of each unique value
```

```
tab <- table(dat$Sepal.Length)
```

```
# Sort highest to lowest.
```

```
sort(tab, decreasing=TRUE)
```

Mode can also be calculated for qualitative variables like Species in this case.

The image shows the RStudio interface. The top menu bar includes File, Edit, Code, View, Plots, Session, Build, Debug, Profile, Tools, and Help. Below the menu is a toolbar with icons for file operations and a search bar labeled 'Go to file/function'. The editor pane shows a script with the following code:

```
1 # Loading iris dataset
2
3 dat <- iris
4
5 # Calculating standard deviation of multiple variables
6
7 lapply(dat[ 1:4 ], sd)
8 _
```

The status bar at the bottom of the editor indicates '8:1 (Top Level)'. Below the editor is a console pane with the following output:

```
R 4.2.1 · ~/
> lapply(dat[ 1:4 ], sd)
$sepal.Length
[1] 0.8280661

$sepal.width
[1] 0.4358663

$petal.Length
[1] 1.765298

$petal.width
[1] 0.7622377
```

Image showing calculation of variance of multiple variables

The screenshot shows the RStudio interface. The source editor contains the following R code:

```
1 # Loading iris dataset
2
3 dat <- iris
4
5 # calculating summary of the dataset
6
7 summary(dat)
8 by(dat, dat$species, summary)
9 -
```

The console output shows the summary of the iris dataset, grouped by species:

```
-----
dat$species: virginica
  Sepal.Length  Sepal.width  Petal.Length  Petal.width
Min.   :4.900    Min.   :2.200   Min.   :4.500   Min.   :1.400
1st Qu.:6.225    1st Qu.:2.800   1st Qu.:5.100   1st Qu.:1.800
Median :6.500    Median :3.000   Median :5.550   Median :2.000
Mean   :6.588    Mean   :2.974   Mean   :5.552   Mean   :2.026
3rd Qu.:6.900    3rd Qu.:3.175   3rd Qu.:5.875   3rd Qu.:2.300
Max.   :7.900    Max.   :3.800   Max.   :6.900   Max.   :2.500
  Species
setosa   : 0
versicolor: 0
virginica :50
```

Image showing summary of the dataset iris

The screenshot shows the RStudio interface. The source editor contains the following R code:

```
1 # Loading iris dataset
2
3 dat <- iris
4
5 # calculating mode the dataset
6
7 tab <- table(dat$sepal.Length)
8
9 # sort from highest to lowest
10 sort(tab, decreasing = TRUE)
11 _
```

The console shows the execution of the code:

```
> tab <- table(dat$sepal.Length)
> # sort from highest to lowest
> sort(tab, decreasing = TRUE)
```

The output is a single row of 18 values representing the sorted frequencies of sepal lengths:

5	5.1	6.3	5.7	6.7	5.5	5.8	6.4	4.9	5.4	5.6	6	6.1	4.8	6.5	4.6	5.2	6.2
---	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	---	-----	-----	-----	-----	-----	-----

The console also shows the following values:

10	9	9	8	8	7	7	7	6	6	6	6	6	5	5	4	4	4
----	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Below these, there is another row of values:

6.9	7.7	4.4	5.9	6.8	7.2	4.7	6.6	4.3	4.5	5.3	7	7.1	7.3	7.4	7.6	7.9	
-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	---	-----	-----	-----	-----	-----	--

And a final row of values:

4	4	3	3	3	3	2	2	1	1	1	1	1	1	1	1	1	
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	--

Image showing mode of a dataset being calculated

```
summary(dat$Species)
```

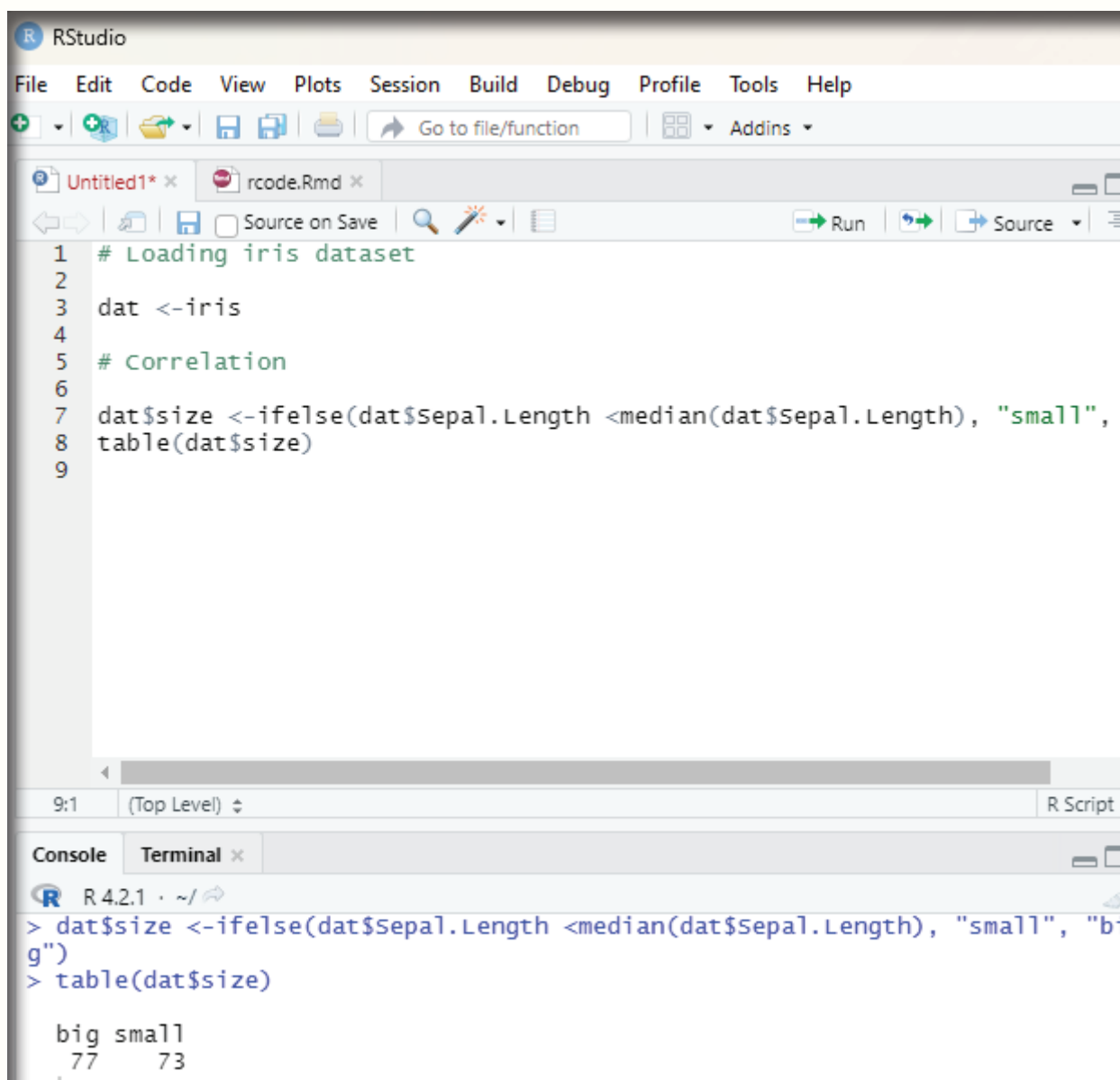
Correlation:

This is another descriptive statistics. This value measures the linear relationship between two variables.

the table() function can be used on two qualitative variables to create a contingency table. The dataset iris has only one qualitative variable so the user needs to create a new qualitative variable for this example. The user can create the variable size which corresponds to small and big.

```
dat$size <- ifelse(dat$Sepal.Length < median(dat$Sepal.Length), "small", "big")
```

```
table(dat$size)
```



The screenshot shows the RStudio interface. The script editor contains the following code:

```
1 # Loading iris dataset
2
3 dat <- iris
4
5 # Correlation
6
7 dat$size <- ifelse(dat$Sepal.Length < median(dat$Sepal.Length), "small",
8 table(dat$size)
9
```

The console output shows the result of the table function:

```
> dat$size <- ifelse(dat$Sepal.Length < median(dat$Sepal.Length), "small", "big")
> table(dat$size)

big small
 77    73
```

Image showing table function in use

One can now create contingency table of two variables Species and size with the table() function.

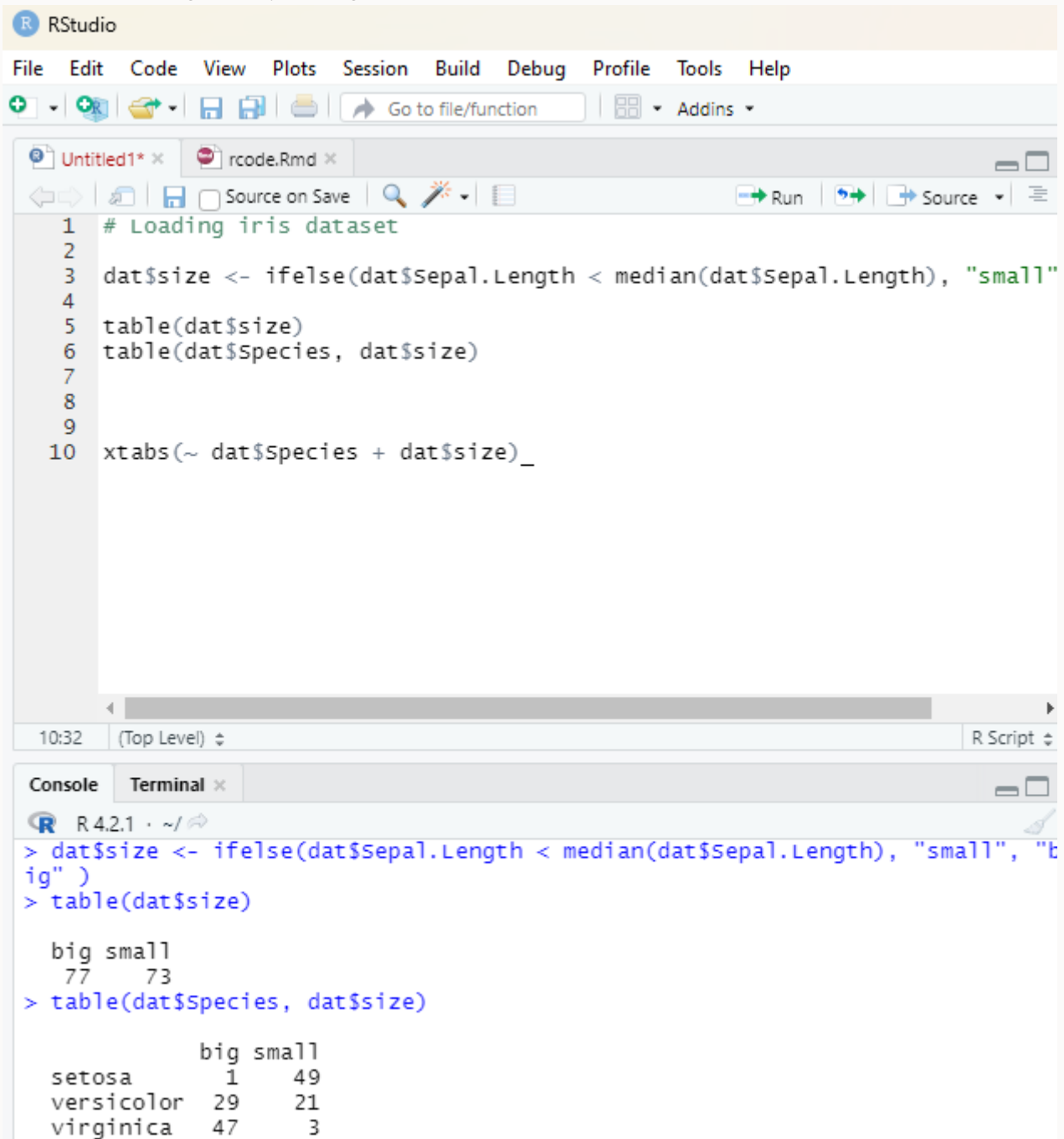
```
dat$size <- ifelse(dat$Sepal.Length < median(dat$Sepal.Length), "small", "big")
```

table(dat\$size)

table(dat\$Species, dat\$size)

xtabs(~ dat\$Species + dat\$size)

Instead of having the frequencies (the actual number of cases) one can also use the relative frequencies (proportions) in each subgroup by adding the table() function inside the prop.table() function.



The screenshot shows the RStudio interface. The script editor contains the following code:

```
1 # Loading iris dataset
2
3 dat$size <- ifelse(dat$Sepal.Length < median(dat$Sepal.Length), "small", "big")
4
5 table(dat$size)
6 table(dat$Species, dat$size)
7
8
9
10 xtabs(~ dat$Species + dat$size)_
```

The console shows the execution of the code:

```
> dat$size <- ifelse(dat$Sepal.Length < median(dat$Sepal.Length), "small", "big")
> table(dat$size)

big small
 77    73
> table(dat$Species, dat$size)

      big small
setosa    1    49
versicolor 29    21
virginica  47     3
```

Image showing xtab function in use

Instead of having the frequencies (the actual number of cases) one can also use the relative frequencies (proportions) in each subgroup by adding the `table()` function inside the `prop.table()` function.

```
prop.table(table(dat$Species, dat$size))
```

Calculating percentages by row:

```
# Round to two digits with round
```

```
round(prop.table(table(dat$Species, dat$size), 1), 2)
```

```
# Calculating percentages by column:
```

```
round(prop.table(table(dat$Species, dat$size), 2), 2)
```

Mosaic plot:

This allows the user to visualize a contingency table of two qualitative variables.

```
mosaicplot(table(dat$Species, dat$size),  
            color = TRUE,  
            xlab = "Species",  
            ylab = "Size")
```

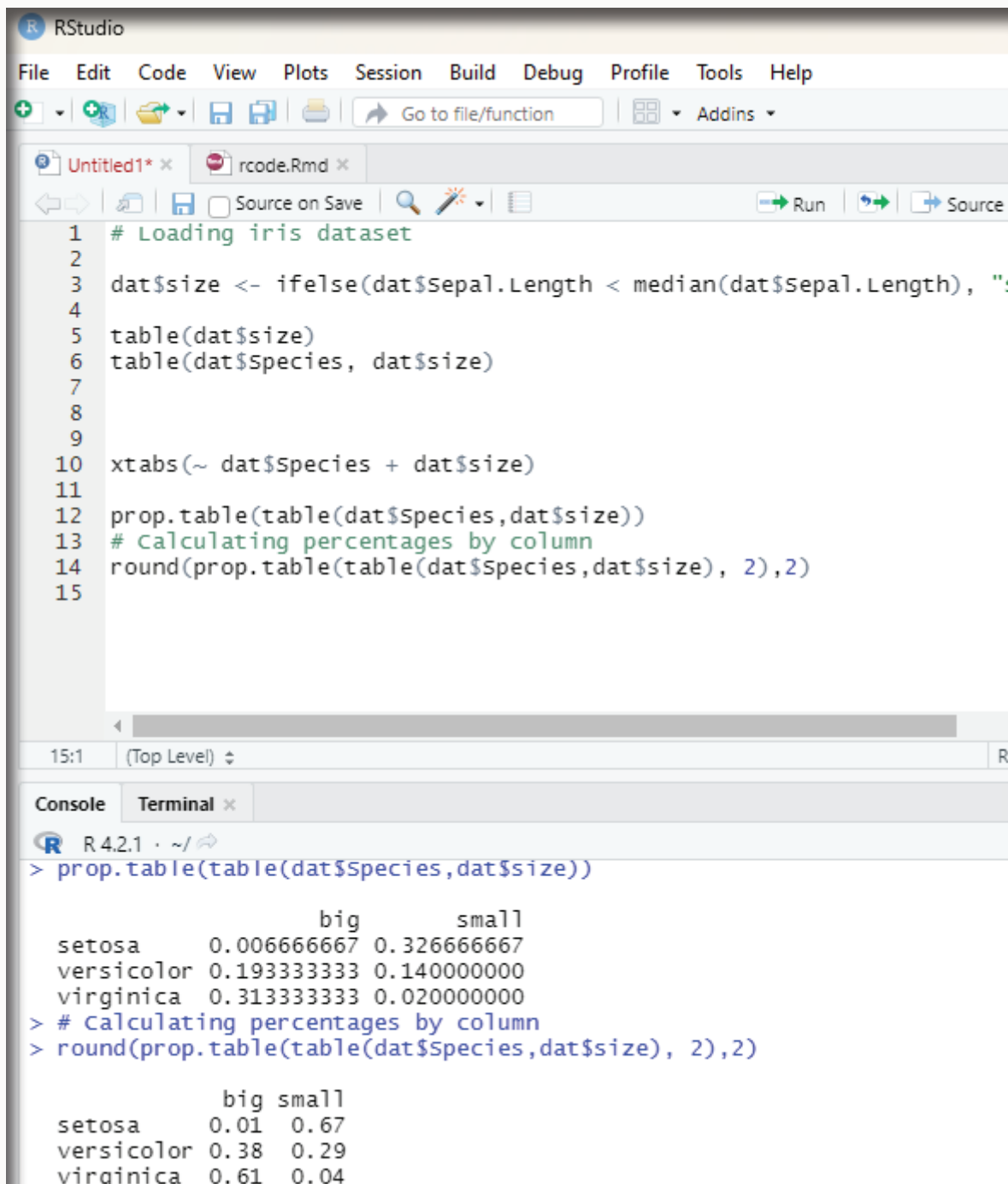
Bar plot:

Bar plots can only be done on qualitative variables. A bar plot is a tool to visualize the distribution of qualitative variable.

```
barplot(table(dat$size))
```

The user can also draw a bar plot of relative frequencies instead of the frequencies by adding `prop.table()`.

```
barplot(prop.table(table(dat$size)))
```



The screenshot shows the RStudio interface with a script editor and a console. The script editor contains the following R code:

```
1 # Loading iris dataset
2
3 dat$size <- ifelse(dat$Sepal.Length < median(dat$Sepal.Length), "small", "big")
4
5 table(dat$size)
6 table(dat$Species, dat$size)
7
8
9
10 xtabs(~ dat$Species + dat$size)
11
12 prop.table(table(dat$Species, dat$size))
13 # calculating percentages by column
14 round(prop.table(table(dat$Species, dat$size), 2), 2)
15
```

The console shows the output of the commands entered:

```
> prop.table(table(dat$Species, dat$size))

      big      small
setosa 0.006666667 0.326666667
versicolor 0.193333333 0.140000000
virginica 0.313333333 0.020000000
> # calculating percentages by column
> round(prop.table(table(dat$Species, dat$size), 2), 2)

      big small
setosa 0.01 0.67
versicolor 0.38 0.29
virginica 0.61 0.04
```

Image showing prop table function in use

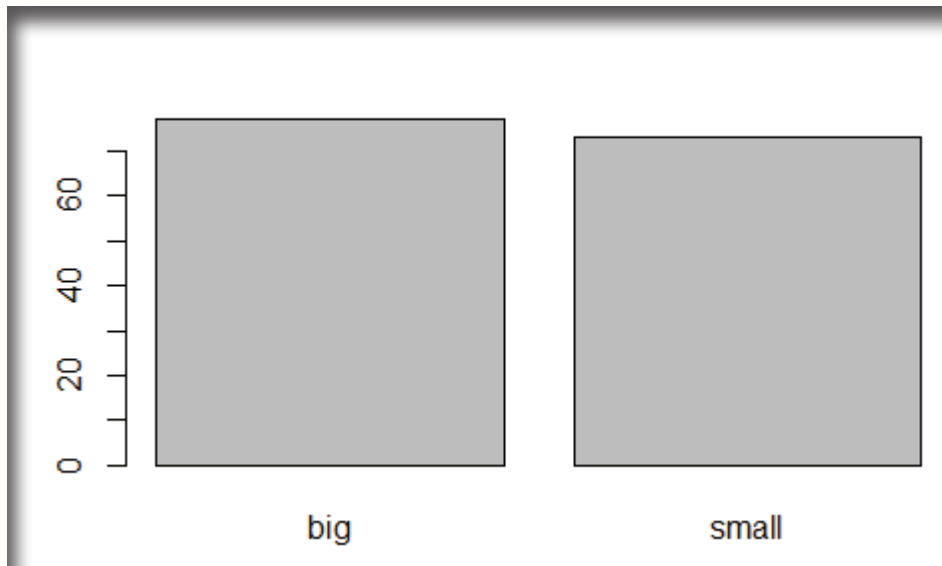


Image showing bar plot

Histogram:

This gives an idea about the distribution of a quantitative variable. The basic idea is to break the ranges of values into intervals and count how many observations fall into each interval.

```
hist()
```

```
hist(dat$Sepal.Length)
```

If the user wants to change the number of bins then the argument `breaks =` is added inside the `hist()`. As a rule of thumb the number of bins should be the rounded value of the square root of the number of observations. This dataset contains 150 observations the number of bins can be set to 12.

```
ggplot(dat) +  
  aes(x = Sepal.Length) +  
  geom_histogram()
```

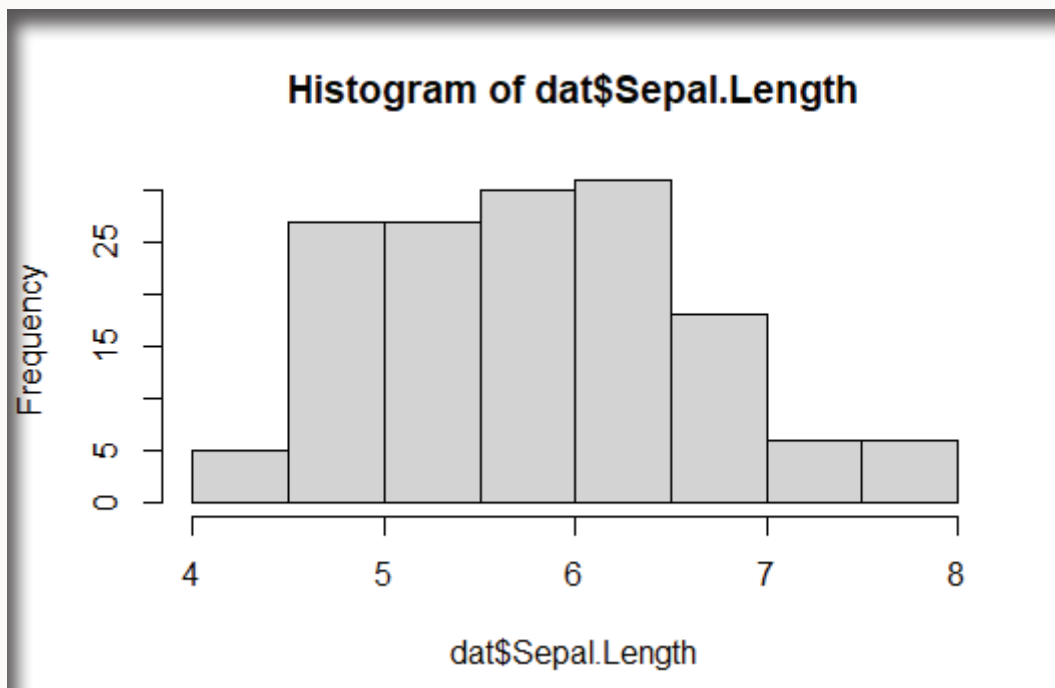


Image showing Histogram created

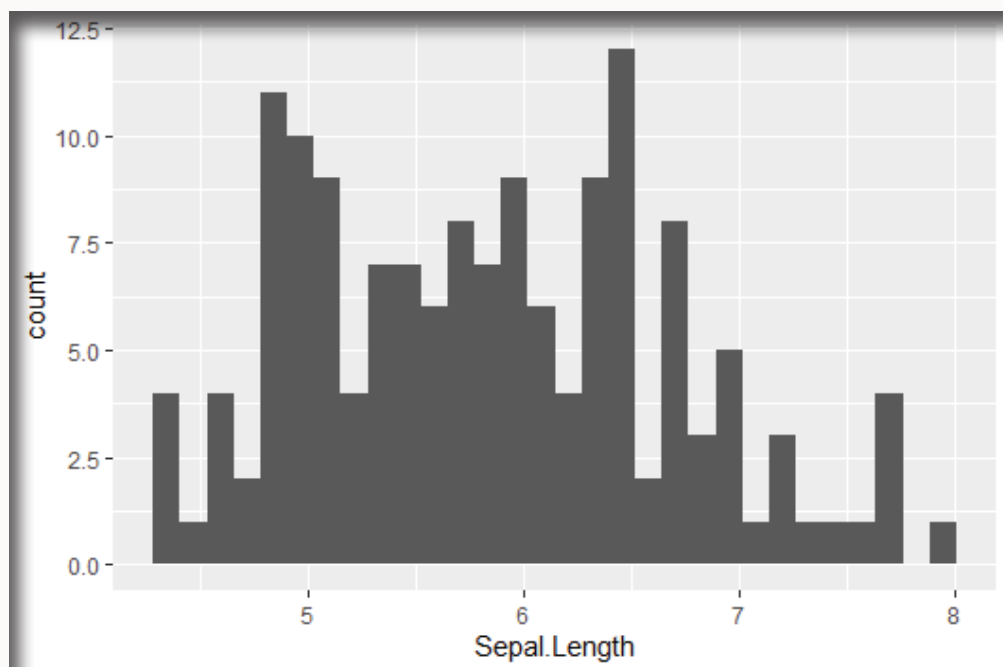


Image showing histogram generated by ggplot

Box plot:

These plots are useful in descriptive statistics. It graphically represents the distribution of a quantitative variable by visually displaying five common location summary (minimum, median, first/third quartiles and maximum) and any observation that was classified as a suspected outlier using the interquartile range criterion.

```
boxplot(dat$Sepal.Length)
```

Dot plot:

This is more or less similar to boxplot except for the fact that observations are represented as points and there is no summary statistics presented on the plot.

```
library(lattice)
```

```
dotplot(dat$Sepal.Length ~ dat$Species)
```

Scatter plot:

This allows the user to check whether there is a potential link between two quantitative variables.

```
plot(dat$Sepal.Length, dat$Petal.Length)
```

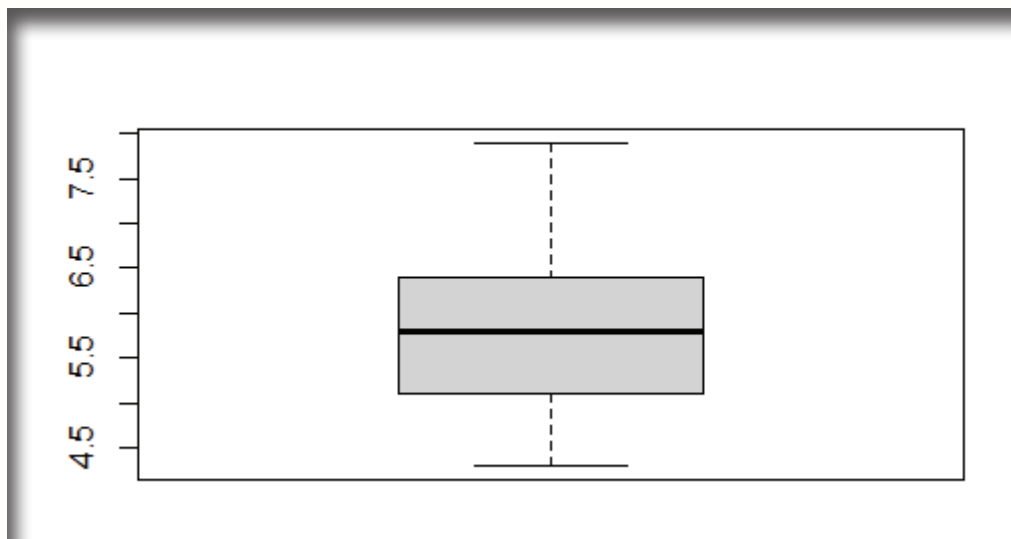


Image showing Box plot generated

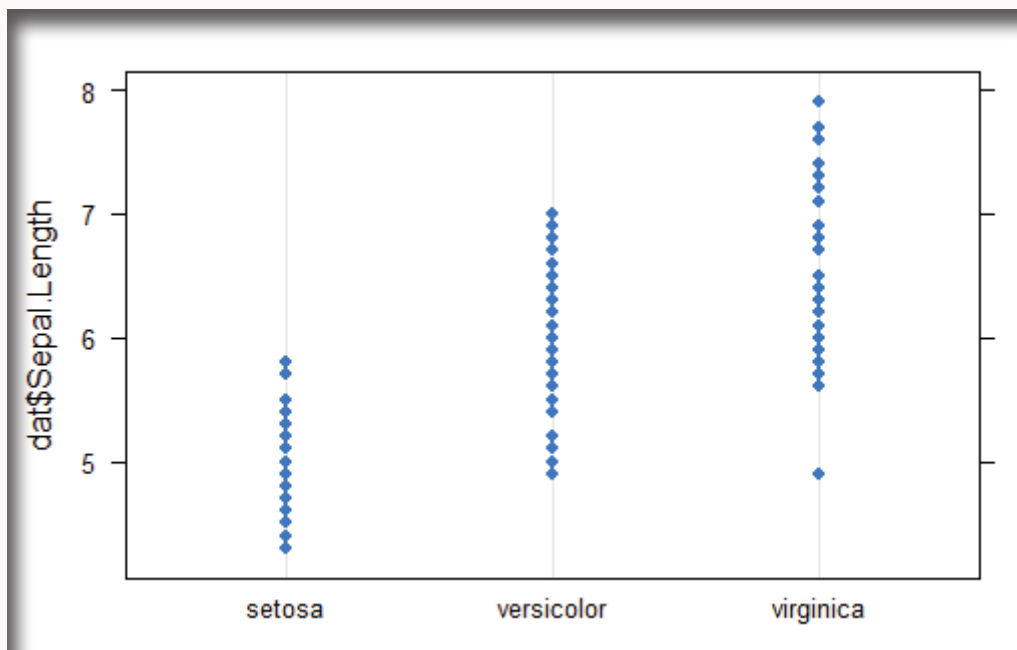


Image showing dot plot generated

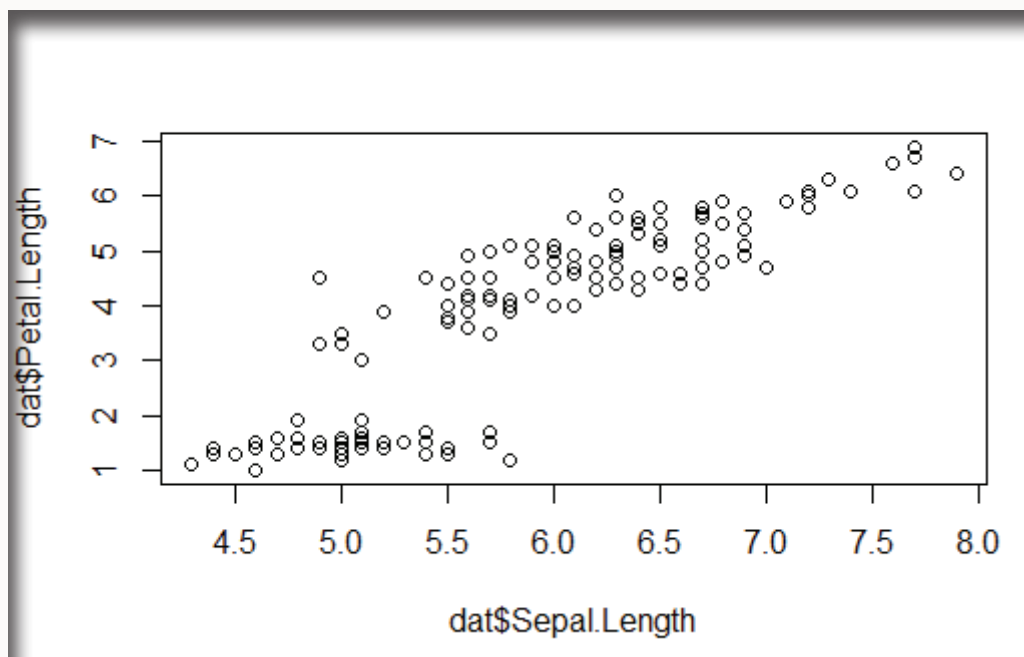


Image showing scatter plot generated

Exploratory Data Analysis

This is actually the very first step in a data project. Exploratory data analysis (EDA) consists of univariate (1-variable) and bivariate (2-variables) analysis. Ideally certain steps need to be followed to lead to the analytic pathway.

Step 1 - First approach to data

Step 2 - Analyzing categorical variables

Step 3 - Analyzing numerical variables

Step 4 - Analyzing numerical and categorical variables at the same time.

Some of the points that are part of basic EDA:

1. Data types
2. Outliers
3. Missing values
4. Distributions (numerically & graphically) for both numerical and categorical variables.

Types of Results EDA provides:

Informative - Plots are classic examples for this type of result. If delivered numerically it would be a long variable summary. Data cannot be filtered out of this summary, but the user can derive lots of information from it.

Operative - The results can be used to take action directly on the data workflow (for example, selecting any variables whose percentage of missing values are below 20%). This type of result is used in the Data Preparation stage.

R packages that are needed to be installed for performing EDA:

1. tidyverse
2. funModeling
3. Hmisc

Code for installing these libraries:

```
install.packages("tidyverse")  
install.packages("funModeling")  
install.packages("Hmisc")
```

The above installed libraries should be loaded first.

Code for loading the libraries:

```
library(funModeling)  
library(tidyverse)  
library(Hmisc)
```

For this purpose the heart_disease data from the funModeling package is used.

Code:

```
data=heart_disease %>% select(age, max_heart_rate, thal, has_heart_disease)  
  
tl:dr (code)  
  
basic_eda <- function(data)  
  {  
    glimpse(data)  
    df_status(data)  
    freq(data)  
    profiling_num(data)  
    plot_num(data)  
    describe(data)  
  }  
  
basic_eda(data)
```

Step 1 - First approach to studying data

Number of observations (rows) and variables, and a head of the first cases are displayed.

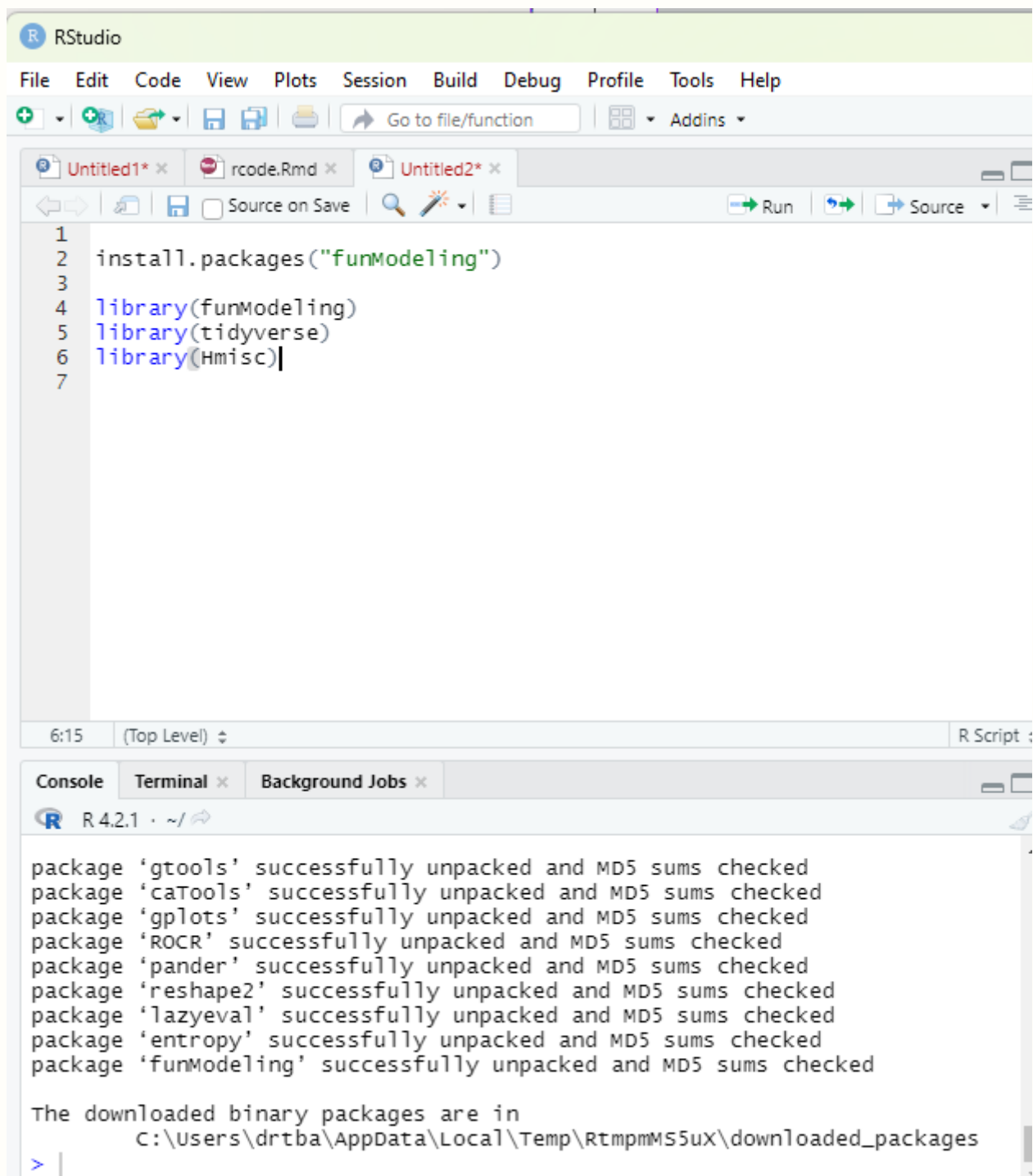


Image showing relevant libraries installed and loaded

Code:

```
glimpse(data)
```

In order to get the metrics about data types, zeros, infinite numbers, and missing values:

```
df_status(data)
```

This code returns a table, so it is easy to keep with variables that match certain conditions like:

Having at least 80% of non-NA values ($p_na < 20$).

Having less than 50 unique values ($unique \leq 50$).

Important of these metrics:

Zeros: Variables containing a large number of zeros may not be useful for modeling and, in some cases they may dramatically bias the model.

NA: Several models automatically exclude rows with NA. Presence of NA in variables would mislead the analysis.

Inf: Infinite values may lead to an unexpected behavior in some R functions.

Type: Some variables are encoded as numbers, but they are codes or categories and the models don't handle them in the same way.

Unique - Factor or categorical variables with a high number of different values tend to do overfitting thereby prevent proper data analysis.

Step 2 - Analyzing categorical variables.

The freq function runs for all factor or character variables automatically.

code:

```
freq(data)
```

Step 3 - Analyzing numerical variables

This can be done graphically.

Code:

```
plot_num(data)
```

Code to analyze variables quantitatively.

```
data_prof=profiling_num(data)
```

The user should bear in mind the following points:

1. Each variable should be described based on their distribution.
2. Attention should be paid to variables with high standard deviation.
3. Selection of metrics that the user is most familiar with.

Analyzing numerical and categorical variables at the same time:

This can be done using Hmisc package.

```
library(Hmisc)  
describe(data)
```

Regression Analysis using R

This is a widely used statistical tool to establish a relationship model between two variables. One of these variables is called the predictor variable whose value is gathered through experiments. The other variable is known as the response variable whose value is derived from the predictor variable.

In Linear Regression these two variables are related through an equation, whose exponent (power) of both these variables is 1. Mathematically, a linear relationship represents a straight line when plotted as a graph. A non-linear relationship is the one in which the exponent of any variable is not equal to 1 and it creates a curve.

General mathematical equation for a linear regression is:

$$y = ax + b$$

y - is the response variable

x - is the predictor variable

a & b - are constants which are also called as coefficients.

Steps to establish a Regression model:

One model that can be considered is attempting to predict weight of a person when the height has been measured. In order to perform this calculation one needs to have a relationship between the height and the weight of the person.

Steps that needs to be followed to create the relationship:

1. Experiment to carry out gathering a sample of observed values of height and corresponding weight.
2. Create a relationship model using the `lm()` functions in R.
3. Find the coefficients from the model created and create a mathematical equation using these.
4. Getting a summary of the relationship model to know the average error in prediction. This is also known as residuals.
5. To predict the weight of new persons, using the `predict()` function in R.

Example:

Input data:

values of height

151, 174, 138, 186, 128, 136, 179, 163, 152, 131

Values in weight

63, 81, 56, 91, 47, 57, 76, 72, 62, 48

Using lm() Function - This function creates a relationship model between the predictor and the response variable.

Syntax:

lm(formula,data)

Formula is a symbol representing the relationship between x and y.

Data is the vector on which the formula will be applied.

Creating relationship model and calculating coefficients:

x <- c(151, 174, 138, 186, 128, 136, 179, 163, 152, 131)

y <- c(63, 81, 56, 91, 47, 57, 76, 72, 62, 48)

Apply the lm() function.

relation <- lm(y~x)

print(relation)

Getting the summary of the relationship:

x <- c(151, 174, 138, 186, 128, 136, 179, 163, 152, 131)

y <- c(63, 81, 56, 91, 47, 57, 76, 72, 62, 48)

Apply the lm() function.

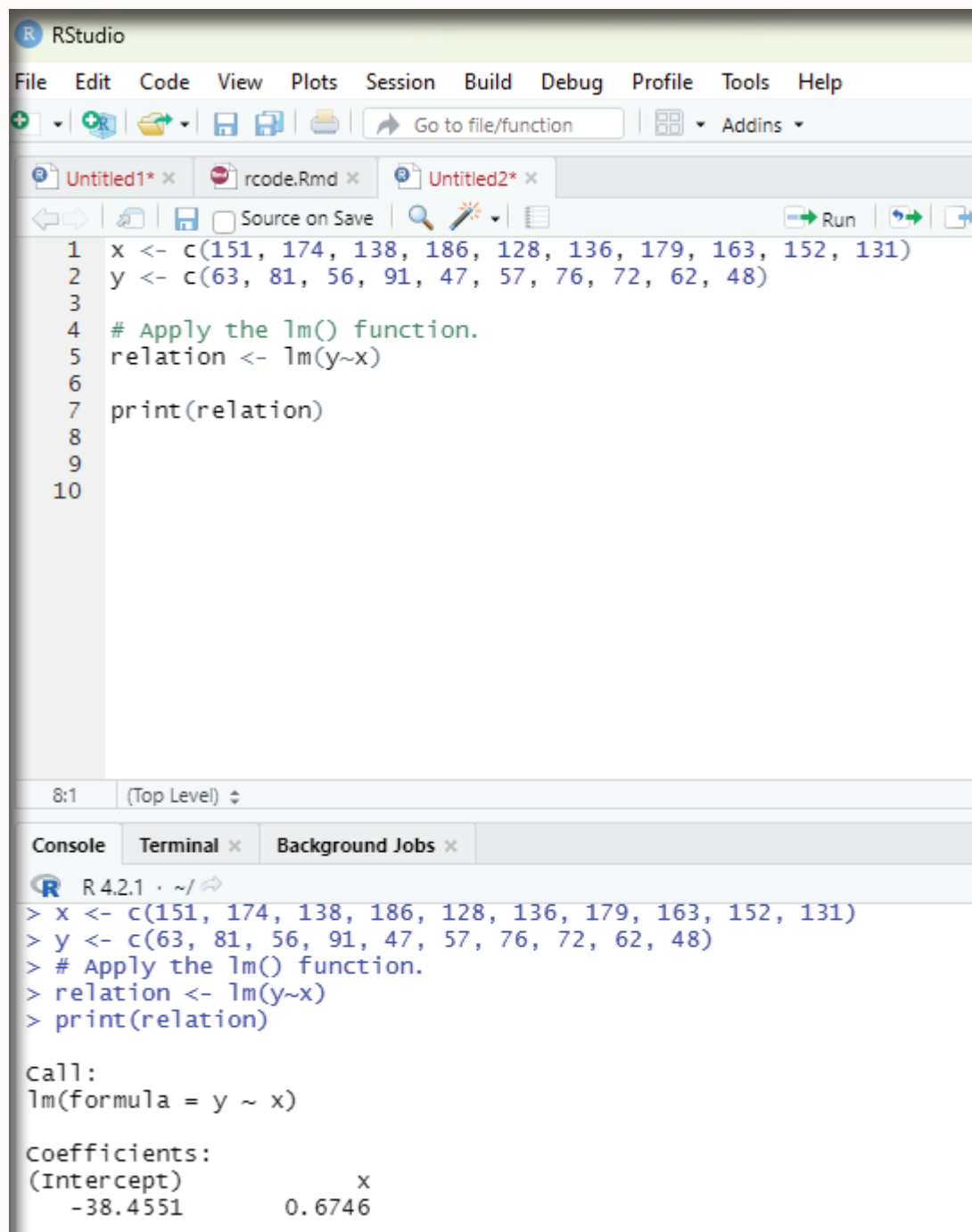
relation <- lm(y~x)

print(summary(relation))

Predict function:

Syntax:

predict(object, newdata)



The image shows the RStudio interface. The top menu bar includes File, Edit, Code, View, Plots, Session, Build, Debug, Profile, Tools, and Help. Below the menu is a toolbar with icons for file operations and a search bar. The main editor window displays a script with the following code:

```
1 x <- c(151, 174, 138, 186, 128, 136, 179, 163, 152, 131)
2 y <- c(63, 81, 56, 91, 47, 57, 76, 72, 62, 48)
3
4 # Apply the lm() function.
5 relation <- lm(y~x)
6
7 print(relation)
8
9
10
```

Below the editor is a console window showing the execution of the code:

```
R 4.2.1 ~ /
> x <- c(151, 174, 138, 186, 128, 136, 179, 163, 152, 131)
> y <- c(63, 81, 56, 91, 47, 57, 76, 72, 62, 48)
> # Apply the lm() function.
> relation <- lm(y~x)
> print(relation)

Call:
lm(formula = y ~ x)

Coefficients:
(Intercept)          x
   -38.4551       0.6746
```

Image showing calculation of coefficients

The screenshot shows the RStudio interface. The script editor contains the following R code:

```
1 x <- c(151, 174, 138, 186, 128, 136, 179, 163, 152, 131)
2 y <- c(63, 81, 56, 91, 47, 57, 76, 72, 62, 48)
3
4 # Apply the lm() function.
5 relation <- lm(y~x)
6
7 print(relation)
8
9 x <- c(151, 174, 138, 186, 128, 136, 179, 163, 152, 131)
10 y <- c(63, 81, 56, 91, 47, 57, 76, 72, 62, 48)
11
12 # Apply the lm() function.
13 relation <- lm(y~x)
14
15 print(summary(relation))
16
17
18
```

The console output shows the results of the linear model:

```
R 4.2.1 ~ /
-6.3002 -1.6629 0.0412 1.8944 3.9775

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) -38.45509    8.04901  -4.778  0.00139 **
x              0.67461    0.05191  12.997 1.16e-06 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 3.253 on 8 degrees of freedom
Multiple R-squared:  0.9548,    Adjusted R-squared:  0.9491
F-statistic: 168.9 on 1 and 8 DF,  p-value: 1.164e-06
```

Image showing summary of relationships calculated

Object is the formula that has been created using `lm()` function.

`newdata` is the vector containing the new value for predictor variable.

```
# The predictor vector.  
x <- c(151, 174, 138, 186, 128, 136, 179, 163, 152, 131)  
  
# The response vector.  
y <- c(63, 81, 56, 91, 47, 57, 76, 72, 62, 48)  
  
# Apply the lm() function.  
relation <- lm(y~x)  
  
# Find weight of a person with height 170.  
a <- data.frame(x = 170)  
result <- predict(relation,a)  
print(result)
```

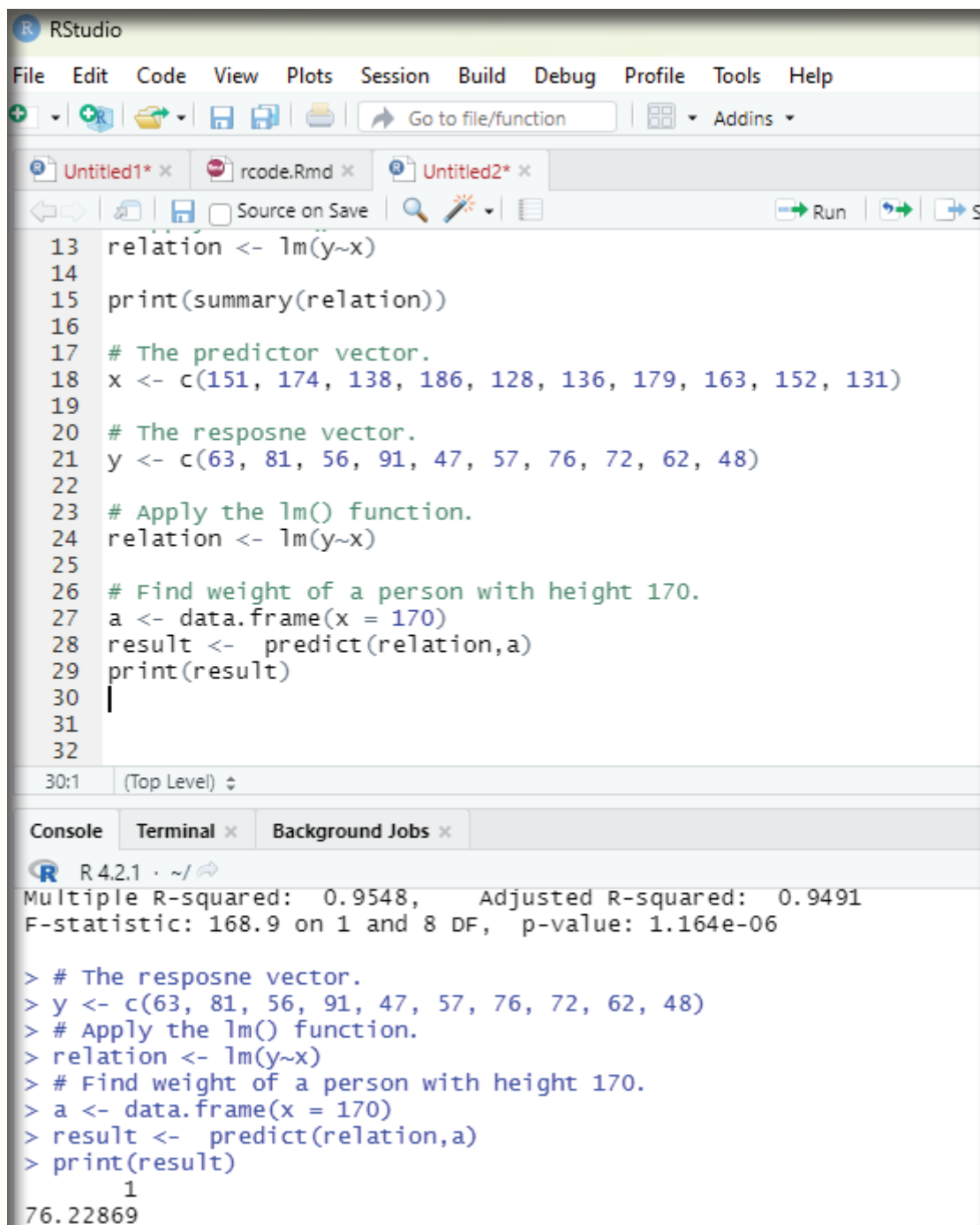
Visualizing the regression graphically:

```
# Create the predictor and response variable.  
x <- c(151, 174, 138, 186, 128, 136, 179, 163, 152, 131)  
y <- c(63, 81, 56, 91, 47, 57, 76, 72, 62, 48)  
relation <- lm(y~x)  
  
# Give the chart file a name.  
png(file = "linearregression.png")  
  
# Plot the chart.  
plot(y,x,col = "blue",main = "Height & Weight Regression",  
abline(lm(x~y)),cex = 1.3,pch = 16,xlab = "Weight in Kg",ylab = "Height in cm")  
  
# Save the file.  
dev.off()
```

Multiple regression using R:

This is an extension of linear regression into relationship between more than two variables. In linear regression there is one predictor and one response variable. In multiple regression there can be more than one predictor variable and one response variable.

In the example database below the comparison should be made between different car models in terms of mileage per gallon (`mpg`), cylinder displacement (`displacement`), horse power (`hp`), weight of the car (`wt`), and some other parameters. The goal should be to establish a relationship between `mpg` as a response to variables like `displacement`, `hp`, and `wt` as predictor variables.

The image is a screenshot of the RStudio interface. The top menu bar includes File, Edit, Code, View, Plots, Session, Build, Debug, Profile, Tools, and Help. Below the menu is a toolbar with icons for file operations and a search bar. The main editor window shows a script with R code. The code defines a linear model 'relation' using 'lm(y~x)', prints its summary, and then uses 'predict()' to find the weight for a height of 170. The console at the bottom shows the output of the commands, including the model's statistics and the predicted weight. The script code is as follows:

```
13 relation <- lm(y~x)
14
15 print(summary(relation))
16
17 # The predictor vector.
18 x <- c(151, 174, 138, 186, 128, 136, 179, 163, 152, 131)
19
20 # The resposne vector.
21 y <- c(63, 81, 56, 91, 47, 57, 76, 72, 62, 48)
22
23 # Apply the lm() function.
24 relation <- lm(y~x)
25
26 # Find weight of a person with height 170.
27 a <- data.frame(x = 170)
28 result <- predict(relation,a)
29 print(result)
30
31
32
```

The console output shows the following:

```
R 4.2.1 ~ /
Multiple R-squared:  0.9548,    Adjusted R-squared:  0.9491
F-statistic: 168.9 on 1 and 8 DF,  p-value: 1.164e-06

> # The resposne vector.
> y <- c(63, 81, 56, 91, 47, 57, 76, 72, 62, 48)
> # Apply the lm() function.
> relation <- lm(y~x)
> # Find weight of a person with height 170.
> a <- data.frame(x = 170)
> result <- predict(relation,a)
> print(result)
      1
76.22869
```

Image showing lm function

Equation for multiple regression:

$$y = a + b_1x_1 + b_2x_2 + \dots + b_nx_n$$

y - response variable

a, b₁, b₂...b_n are coefficients

x₁, x₂, ...x_n are predictor variables

Regression model can be created using lm() function in R.

lm() function creates a relationship model between the predictor and response variable.

Syntax:

lm(y ~ x₁+x₂+x₃..., data)

Using mtcars database:

```
input <- mtcars[,c("mpg","disp","hp","wt")]
print(head(input))
```

Creating a relationship model and get coefficients:

```
input <- mtcars[,c("mpg","disp","hp","wt")]

# Create the relationship model.
model <- lm(mpg~disp+hp+wt, data = input)

# Show the model.
print(model)

# Get the Intercept and coefficients as vector elements.
cat("# # # # The Coefficient Values # # # ", "\n")

a <- coef(model)[1]
print(a)

Xdisp <- coef(model)[2]
Xhp <- coef(model)[3]
Xwt <- coef(model)[4]

print(Xdisp)
print(Xhp)
print(Xwt)
```

Creating equation for Regression Model:

Based on the above intercept and coefficient values a mathematical equation can be created.

$$Y = a + X_{\text{disp}}.x_1 + X_{\text{hp}}.x_2 + X_{\text{wt}}.x_3$$

or

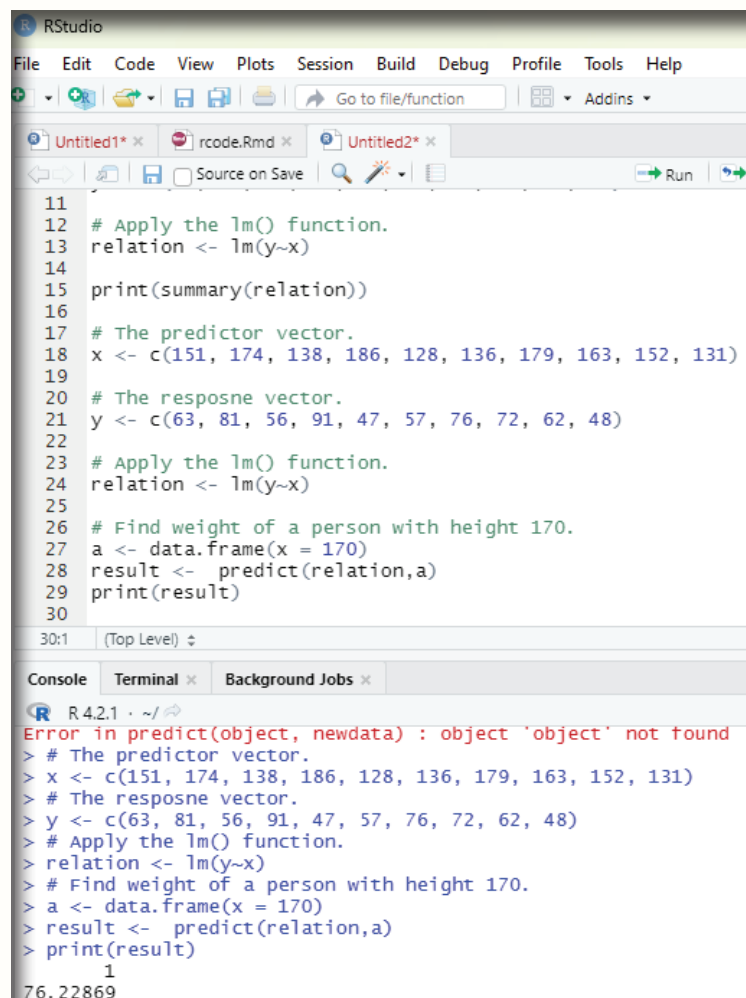
$$Y = 37.15 + (-0.000937) \cdot x_1 + (-0.0311) \cdot x_2 + (-3.8008) \cdot x_3$$

Applying Equation for predicting new values:

One can use the Regression equation created above to predict the mileage when a new set of values for displacement, horse power and weight is provided.

For a car with disp = 300, hp = 100, and wt = 3 the predicted mileage is _____.

$$Y = 37.15 + (-0.000937) \cdot 300 + (-0.0311) \cdot 100 + (-3.8008) \cdot 3 = 22.7104$$



```
11
12 # Apply the lm() function.
13 relation <- lm(y~x)
14
15 print(summary(relation))
16
17 # The predictor vector.
18 x <- c(151, 174, 138, 186, 128, 136, 179, 163, 152, 131)
19
20 # The resposne vector.
21 y <- c(63, 81, 56, 91, 47, 57, 76, 72, 62, 48)
22
23 # Apply the lm() function.
24 relation <- lm(y~x)
25
26 # Find weight of a person with height 170.
27 a <- data.frame(x = 170)
28 result <- predict(relation,a)
29 print(result)
30
```

Console

```
R 4.2.1 ~/  
Error in predict(object, newdata) : object 'object' not found  
> # The predictor vector.  
> x <- c(151, 174, 138, 186, 128, 136, 179, 163, 152, 131)  
> # The resposne vector.  
> y <- c(63, 81, 56, 91, 47, 57, 76, 72, 62, 48)  
> # Apply the lm() function.  
> relation <- lm(y~x)  
> # Find weight of a person with height 170.  
> a <- data.frame(x = 170)  
> result <- predict(relation,a)  
> print(result)  
1  
76.22869
```

Image showing the results of predict function

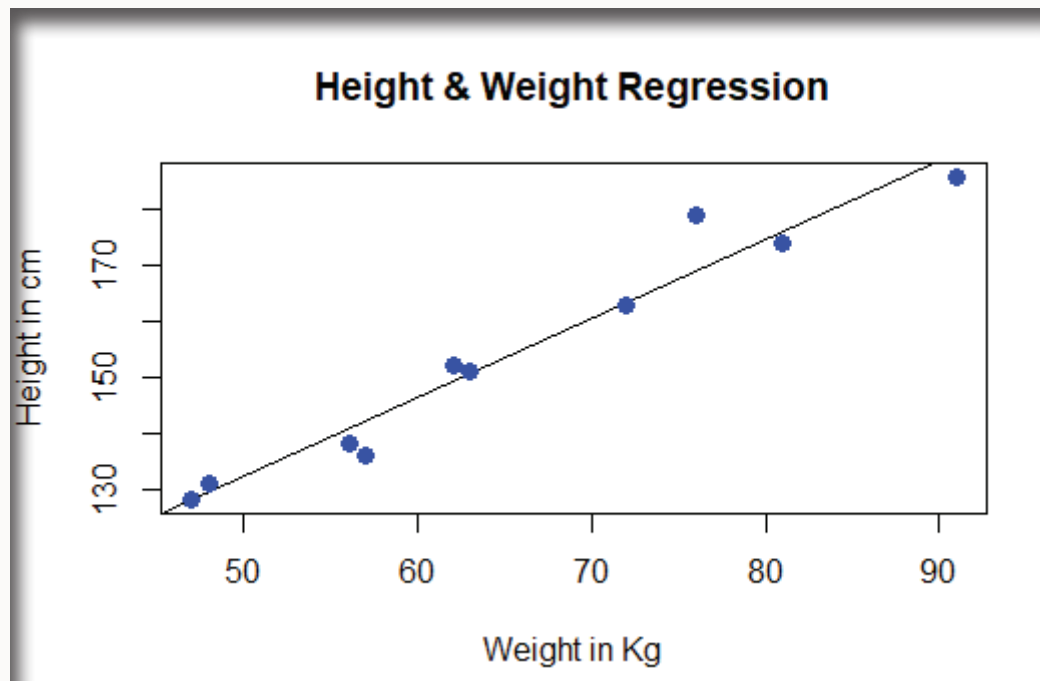


Image showing regression chart

R Charts and Graphs

R language has a number of libraries that can be used to create charts and graphs. Charts and graphs are integral parts of data analysis.

Pie chart:

This is a representation of values as slices of a circle with different colors. The slices are labeled and the numbers corresponding to each slice is also represented in the chart. Pie chart can be created using `pie()` function. Additional parameters can be used to control labels, color title etc.

Syntax:

```
pie(x, labels, radius, main, col, clockwise)
```

x - Is a vector containing the numeric values used in the pie chart.

Labels - Is used to give description to the slices.

Radius - Indicates the radius of the circle of the pie chart (value between -1 and +1).

Main - Indicates the title of the chart.

Col - Indicates the color palette.

Clockwise - Is a logical value indicating if the slices are drawn clockwise or anticlockwise.

Example:

```
# Create data for the graph
```

```
x <- c(24, 45, 18, 56)
```

```
labels <- c("Chennai", "Calcutta", "Trivandrum", "Delhi")
```

```
# Give the chart file a name
```

```
png(file = "city.png")
```

```
# Plot the chart.
```

```
pie(x, labels)
```

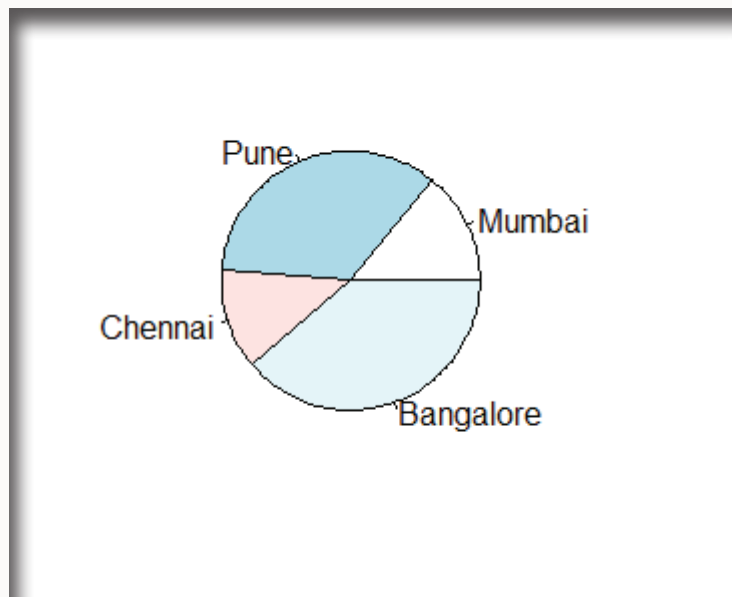
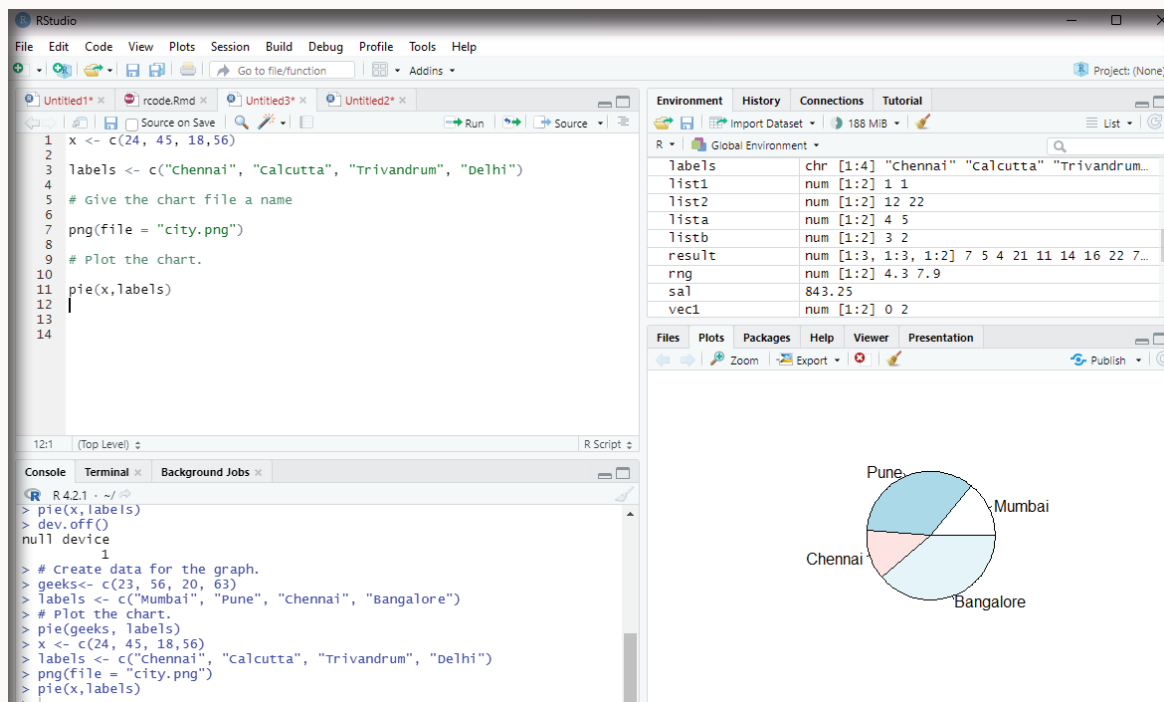


Image showing pie chart generated



Pie Chart Title and colors:

The user can expand the features of the chart by adding more parameters to the function. The parameter `main` is used to add a title to the chart and another parameter `col` will use rainbow color pallet while drawing the chart. The length of the pallet should ideally be the same as the number of values the user has for the chart and hence `length(x)` is used.

Example:

```
# Create data for the graph

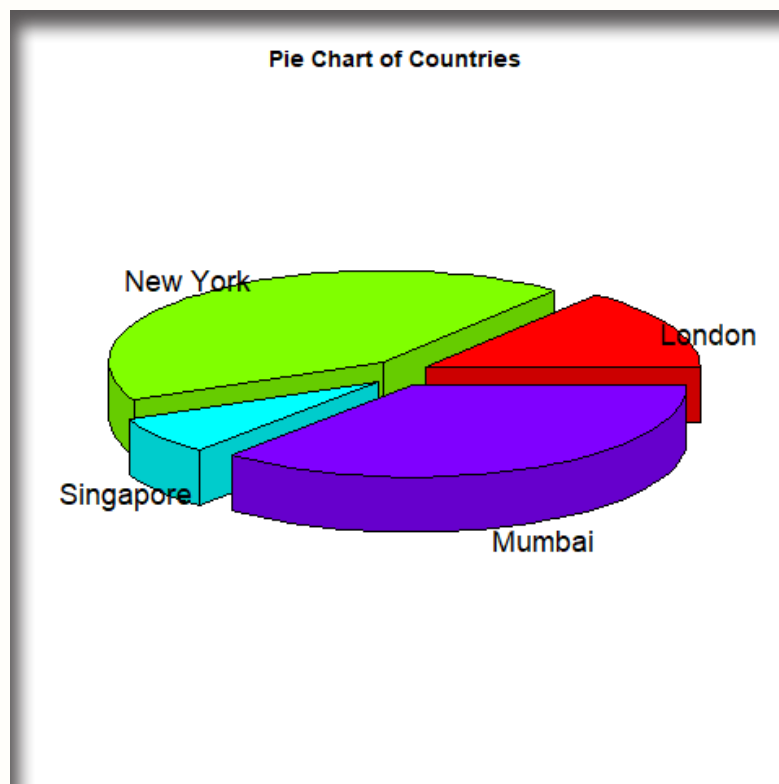
x <- c(32, 45, 76, 52)
labels <- c("Mumbai", "Delhi", "Chennai", "Calcutta")

# Giving the chart a name

png(file= "city_title_colours.jpg")

# Plotting the chart wiht title and rainbow pallet.

pie(x, labels, main = "City pie chart", col = rainbow(length(x)))
```



```

# Create data for the graph.
x <- c(21, 62, 10, 53)
labels <- c("London", "New York", "Singapore", "Mumbai")

piepercent <- round(100*x/sum(x), 1)

# Give the chart file a name.
png(file = "city_percentage_legends.jpg")

# Plot the chart.
pie(x, labels = piepercent, main = "City pie chart", col = rainbow(length(x)))
legend("topright", c("London", "New York", "Singapore", "Mumbai"), cex = 0.8,
      fill = rainbow(length(x)))

# Save the file.
dev.off()

```

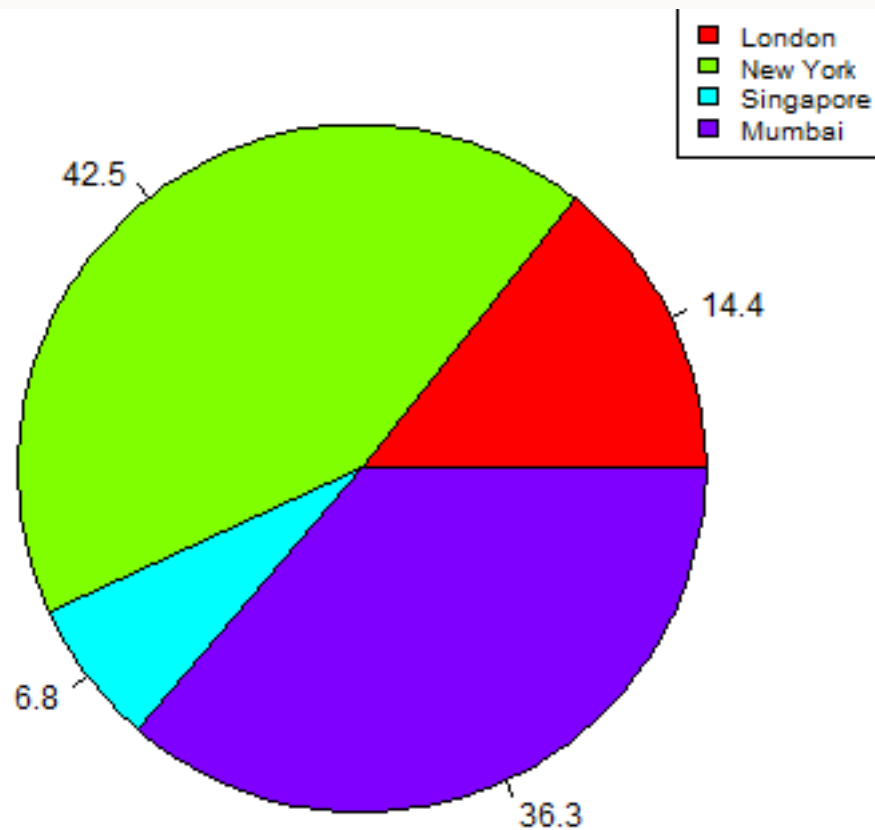


Image showing pie chart with rainbow colors created

The user should bear in mind that the pie chart created will be stored within the working folder.

Bar plot:

This chart represents data in rectangular bars with length of the bar proportional to the value of the variable. R uses the function `barplot()` to create bar charts. R can display both vertical and horizontal bars in the bar chart. In bar chart each of the bars can be given different colors.

Basic syntax to create bar-chart:

```
barplot(H,xlab,ylab,main, names.arg,col)
```

H - vector or matrix containing numeric values used in the bar chart.

xlab - is the label for x axis.

ylab - is the label for y axis.

main - is the title of the bar chart.

names.arg - is a vector of names appearing under each bar.

col - is used to give colors to the bars in the graph.

Example:

```
# Creating data for the chart
```

```
S <- c(14,28,62,83,90)
```

```
# Providing the chart file with a name  
png(file = "barchart.png")
```

```
# Plot the chart
```

```
barplot(S)
```

```
# Saving the file  
dev.off()
```

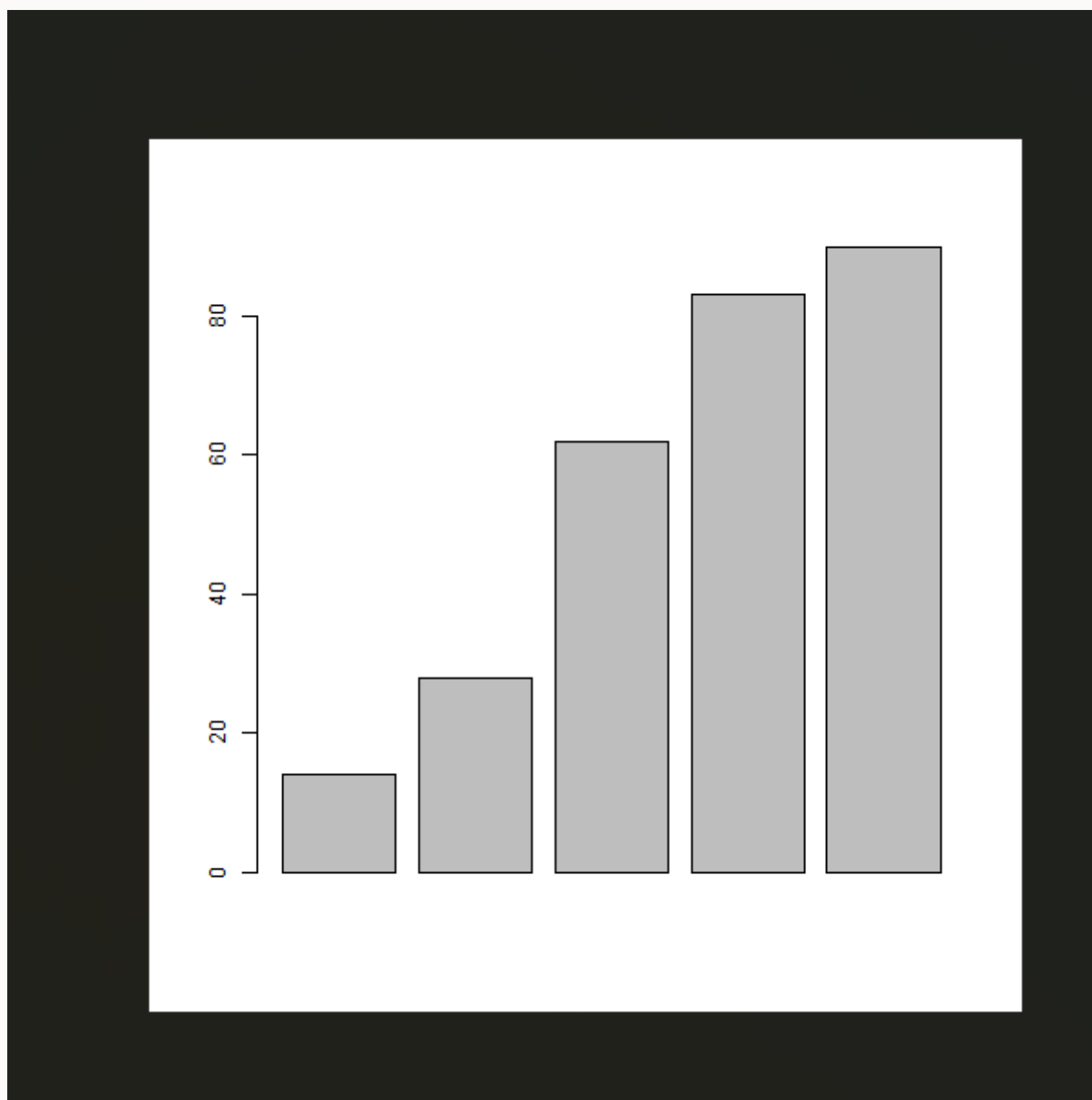


Image showing bar-chart

The image shows the RStudio interface. The top menu bar includes File, Edit, Code, View, Plots, Session, Build, Debug, Profile, Tools, and Help. Below the menu is a toolbar with icons for file operations and a search bar. The editor pane shows a script with the following R code:

```
1 # Creating data for the chart
2
3 s <- c(14,28,62,83,90)
4
5 # Providing the chart file with a name
6 png(file = "barchart.png")
7
8 # Plot the chart
9
10 barplot(s)
11
12 # saving the file
13 dev.off()
14
```

The console pane at the bottom shows the execution output:

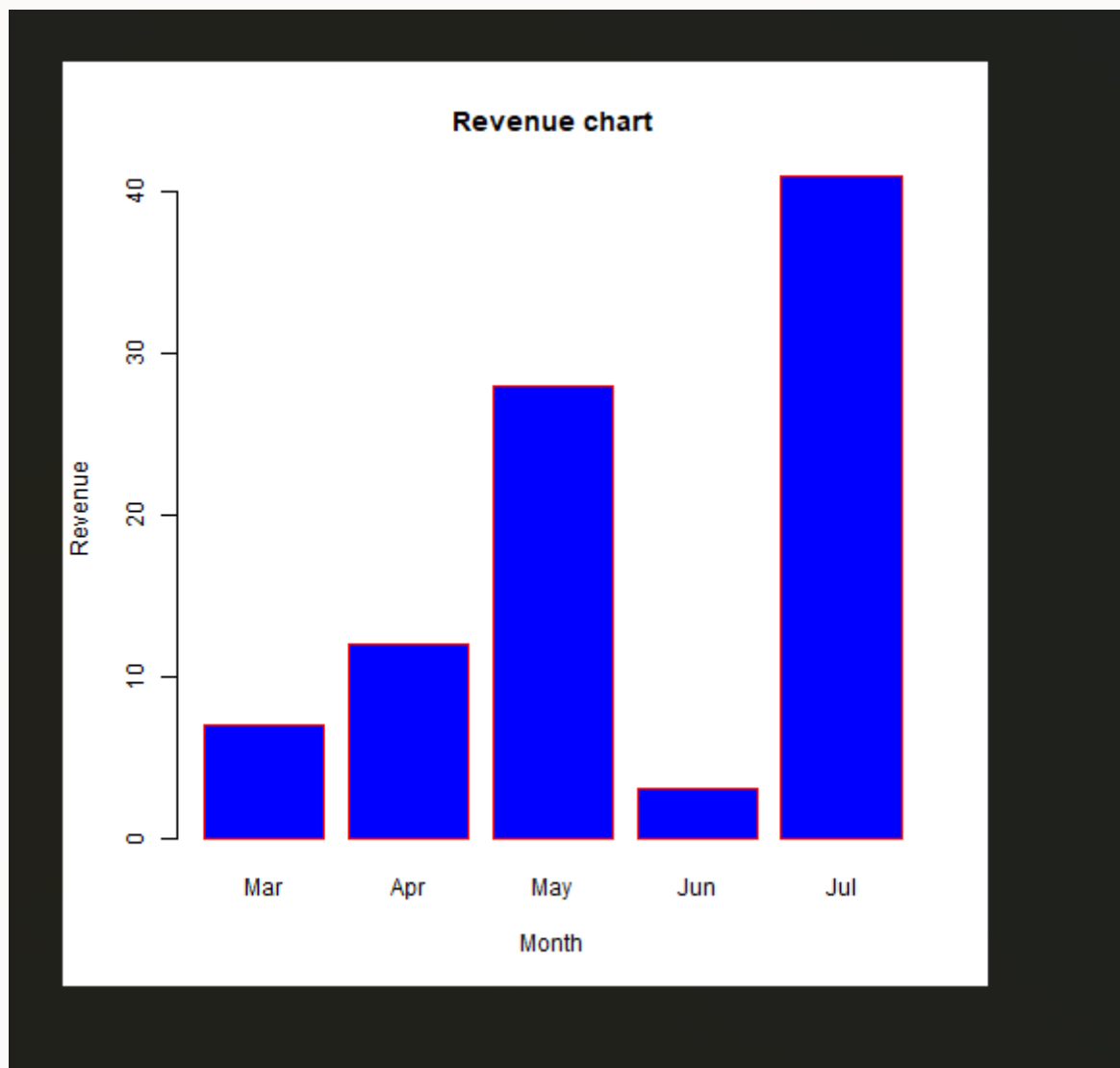
```
R 4.2.1 ~/>
> s <- c(14,28,62,83,90)
> # Providing the chart file with a name
> png(file = "barchart.png")
> barplot(s)
> # saving the file
> dev.off()
null device
      1
>
```

Image showing code executed to create a bar chart

Adding labels, colors and Titles to bar charts:

Example:

```
# Create the data for the chart  
H <- c(7,12,28,3,41)  
M <- c("Mar","Apr","May","Jun","Jul")  
  
# Give the chart file a name  
png(file = "barchart_months_revenue.png")  
  
# Plot the bar chart  
barplot(H,names.arg=M,xlab="Month",ylab="Revenue",col="blue",  
main="Revenue chart",border="red")  
  
# Save the file  
dev.off()
```



Bar chart colored created

Group bar chart and stacked bar chart:

One can create bar chart with groups of bars and stacks in each bar by using a matrix of input values.

```
# Create the input vectors.
colors = c("green","orange","brown")
months <- c("Mar","Apr","May","Jun","Jul")
regions <- c("East","West","North")

# Create the matrix of the values.
Values <- matrix(c(2,9,3,11,9,4,8,7,3,12,5,2,8,10,11), nrow = 3, ncol = 5, byrow = TRUE)

# Give the chart file a name
png(file = "barchart_stacked.png")

# Create the bar chart
barplot(Values, main = "total revenue", names.arg = months, xlab = "month", ylab = "revenue",
        col = colors)

# Add the legend to the chart
legend("topleft", regions, cex = 1.3, fill = colors)

# Save the file
dev.off()
```

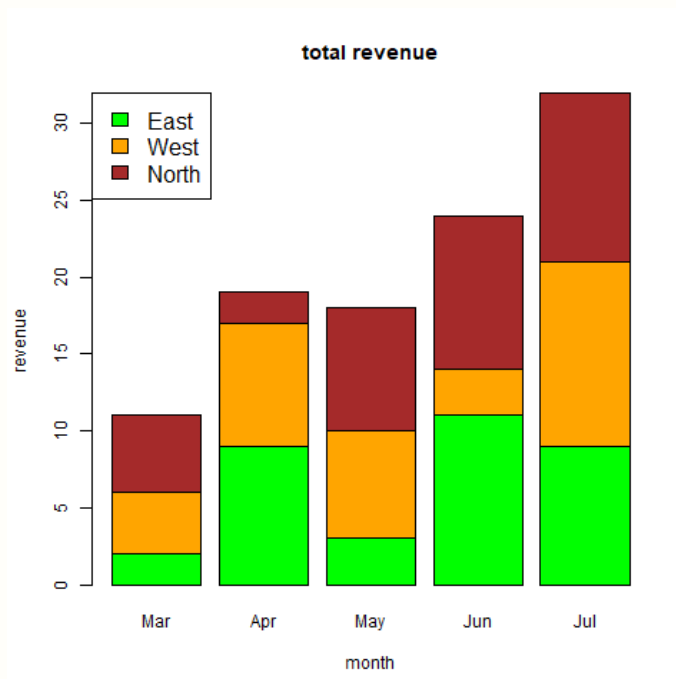


Image showing stacked bar chart

Boxplots:

This is a measure of how well distributed the data in a dataset is. It divides the dataset into three quartiles. The graph represents the minimum, maximum, median, first quartile, the third quartile in the dataset. It is also useful in comparing the distribution of data across datasets by drawing boxplots for each of them.

Boxplots are created using `boxplot()` function.

Syntax:

```
boxplot(x, data, notch, varwidth, names, main)
```

`x` - is a vector or a formula

`data` - is the data frame

`notch` - is a logical value. Set as TRUE to draw a notch.

`varwidth` - is a logical value. Set as TRUE to draw width of the box proportionate to the sample size.

`names` - are the group labels which will be printed under each boxplot.

`main` - is used to give the title of the graph

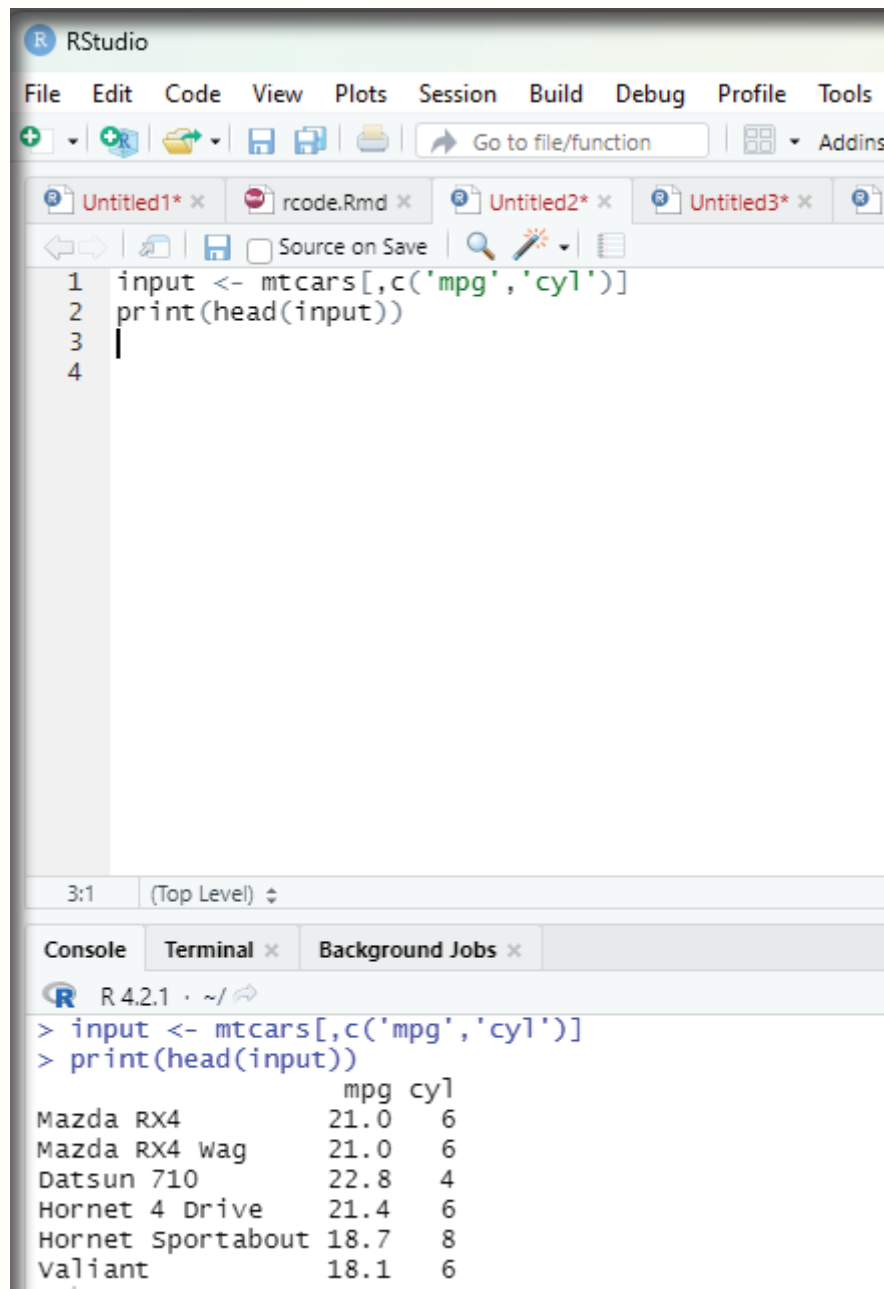
Example:

Dataset "mtcars" which is available in R environment is used.

```
input <- mtcars[,c('mpg','cyl')]  
print(head(input))
```

This code on execution produces the following output:

```
      mpg  cyl  
Mazda RX4    21.0  6  
Mazda RX4 Wag 21.0  6  
Datsun 710    22.8  4  
Hornet 4 Drive 21.4  6  
Hornet Sportabout 18.7  8  
Valiant      18.1  6
```



The image shows the RStudio interface. The source editor contains the following R code:

```
1 input <- mtcars[,c('mpg','cyl')]
2 print(head(input))
3
4
```

The console shows the execution of these commands, resulting in a table of the first six rows of the 'mtcars' dataset, specifically the 'mpg' and 'cyl' columns.

```
> input <- mtcars[,c('mpg','cyl')]
> print(head(input))
```

	mpg	cyl
Mazda RX4	21.0	6
Mazda RX4 wag	21.0	6
Datsun 710	22.8	4
Hornet 4 Drive	21.4	6
Hornet Sportabout	18.7	8
valiant	18.1	6

Image showing result of input command

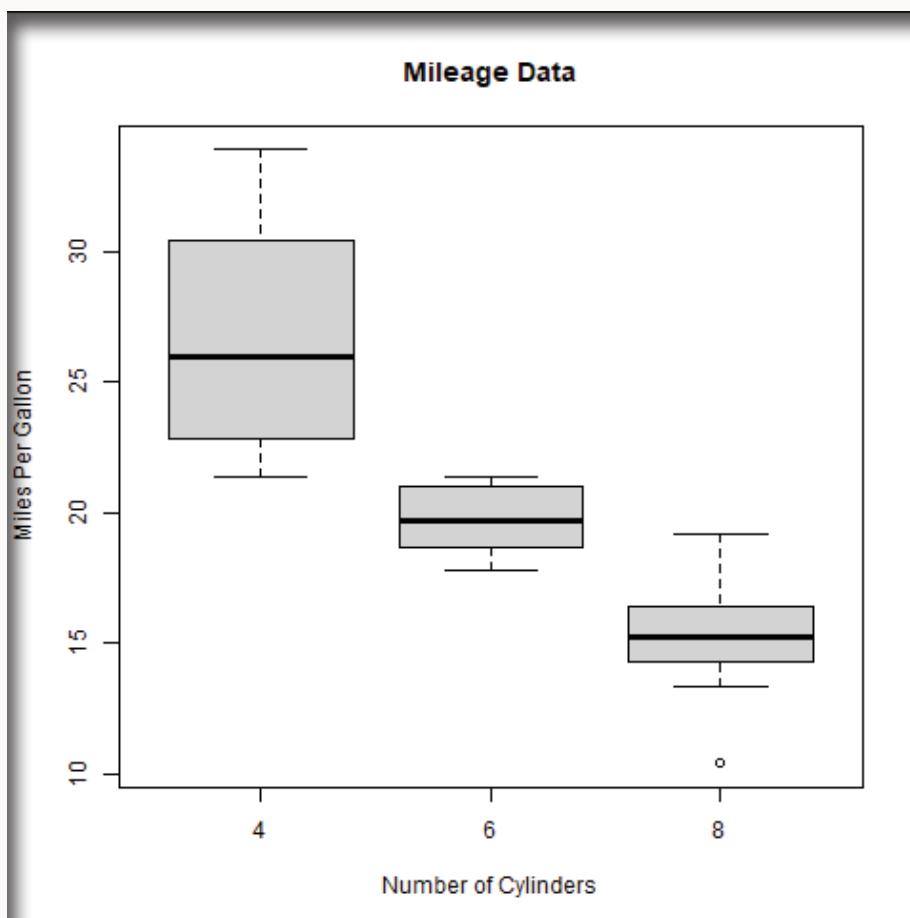


Image showing boxplot generated

Creating the boxplot:

```
# Give the chart file a name.  
png(file = "boxplot.png")
```

```
# Plot the chart.  
boxplot(mpg ~ cyl, data = mtcars, xlab = "Number of Cylinders",  
ylab = "Miles Per Gallon", main = "Mileage Data")
```

```
# Save the file.  
dev.off()
```

Creating boxplot with notch:

This type of boxplot helps the user to find out how the medians of different data groups match with each other.

```
# Give the chart file a name.  
png(file = "boxplot_with_notch.png")  
  
# Plot the chart.  
boxplot(mpg ~ cyl, data = mtcars,  
        xlab = "Number of Cylinders",  
        ylab = "Miles Per Gallon",  
        main = "Mileage Data",  
        notch = TRUE,  
        varwidth = TRUE,  
        col = c("green", "yellow", "purple"),  
        names = c("High", "Medium", "Low")  
        )  
# Save the file.  
dev.off()
```

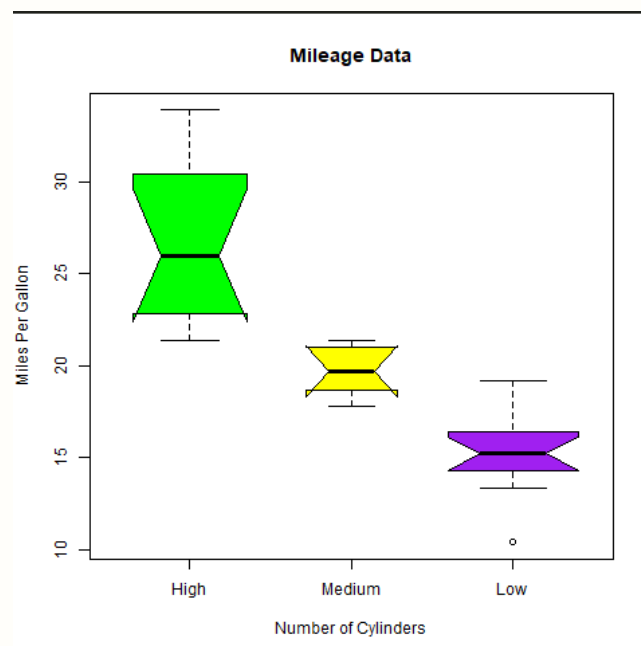


Image showing boxplot with notch created

Histogram:

This represents the frequencies of values of a variable bucketed into ranges. This is similar to that of bar chart, but the difference being that it groups values into continuous ranges. Each bar in histogram represents the height of the number of values present in that range.

Histogram can be generated in R using hist() function. It takes the vector as an input and uses some more parameters to plot it.

Syntax:

```
hist(v,main,xlab,xlim,ylim,breaks,col, border)
```

v - is a vector containing numeric values used in histogram.

main - indicates the title of the chart.

col - is used to set color of the bars.

border - is used to set border color of each bar.

xlab - is used to give description of x-axis.

xlim - is used to specify the range of values on the x-axis.

ylim - is used to specify the range of values on the y-axis.

breaks - is used to mention the width of each bar.

Example:

```
# Create data for the graph
```

```
v <- c(8,18,28,7,42,28,18,56,38,43,18)
```

```
# Give the chart file a name.
```

```
png(file = "histogram.png")
```

```
# Create the histogram
```

```
hist(v,xlab = "Weight",col = "yellow",border = "blue")
```

```
# Save the file
```

```
dev.off()
```

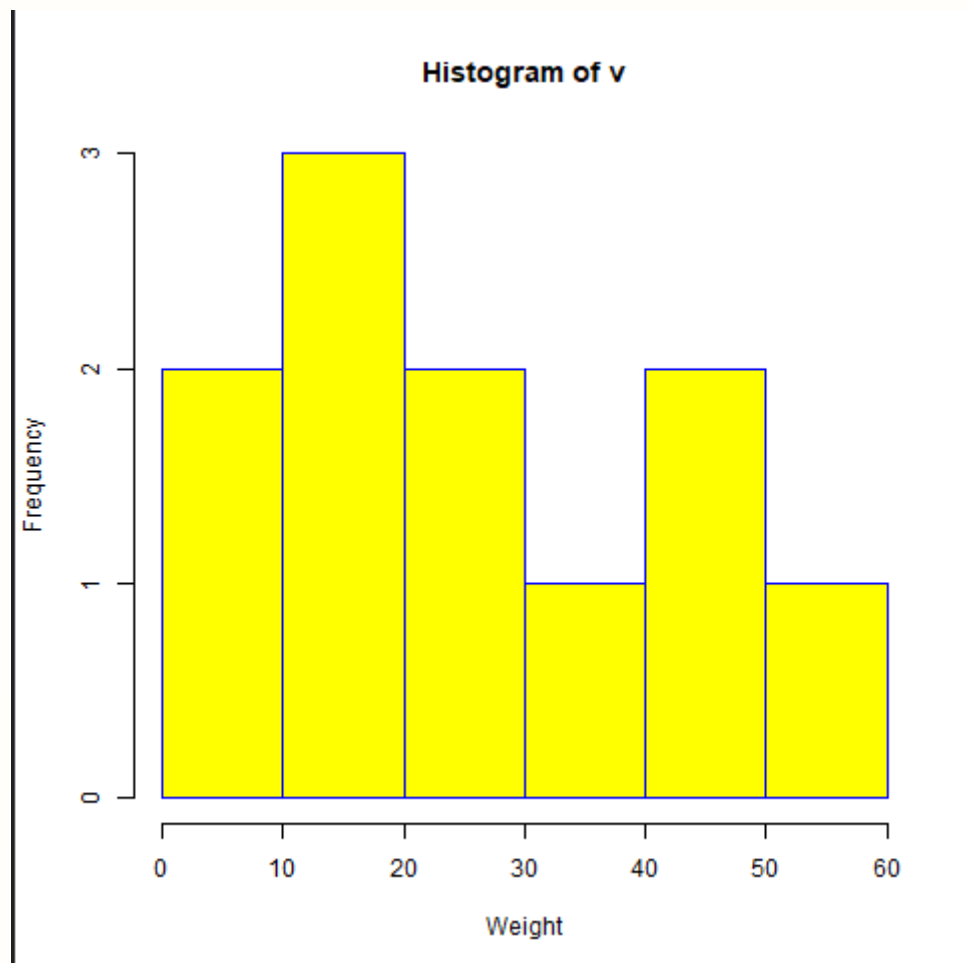
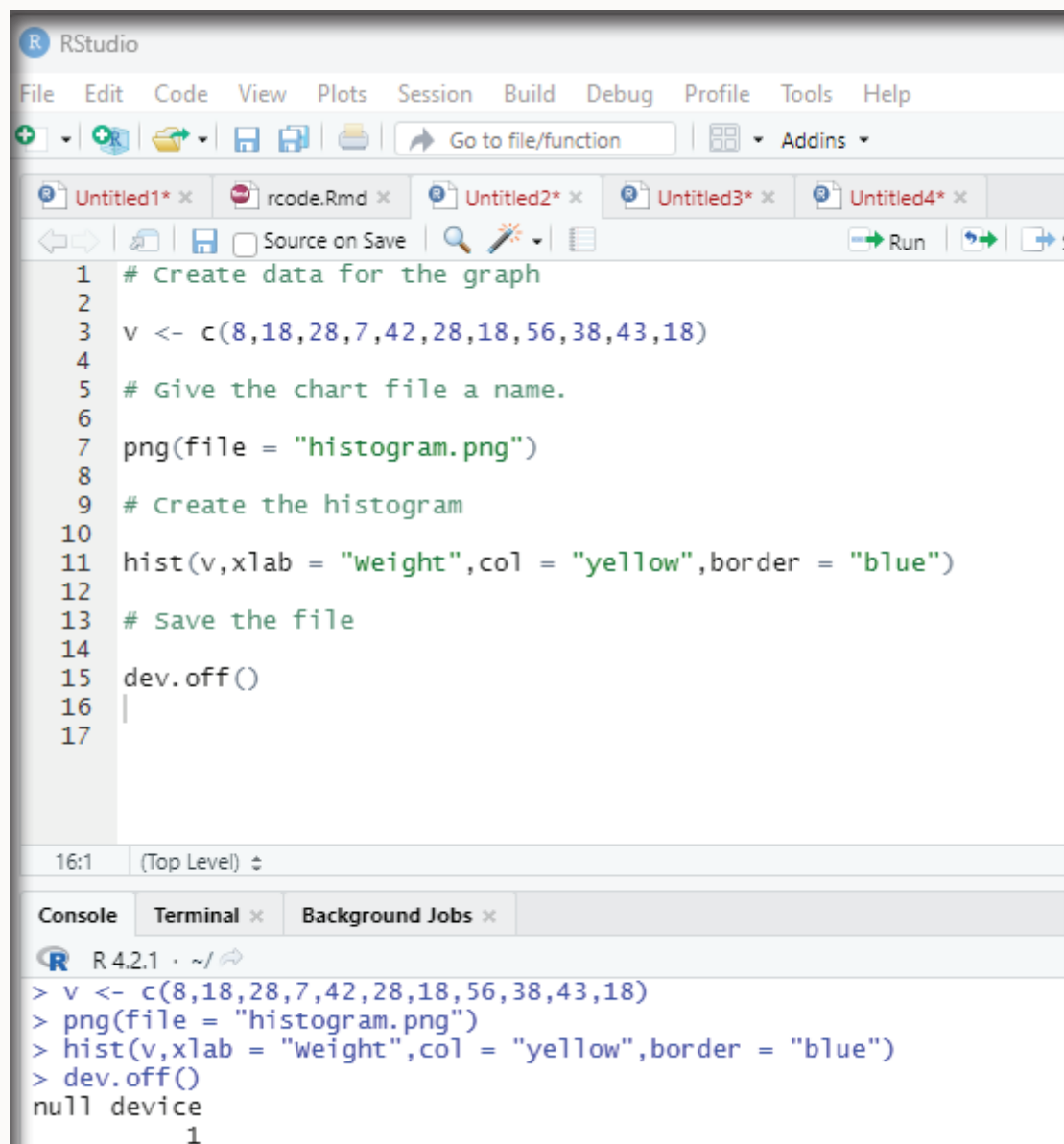


Image showing the histogram generated. This will be saved in the working folder. The user needs to open up the working folder to access this file.

The image shows the RStudio interface. The top menu bar includes File, Edit, Code, View, Plots, Session, Build, Debug, Profile, Tools, and Help. Below the menu is a toolbar with icons for file operations and a search bar labeled 'Go to file/function'. The editor pane shows a script with the following R code:

```
1 # Create data for the graph
2
3 v <- c(8,18,28,7,42,28,18,56,38,43,18)
4
5 # Give the chart file a name.
6
7 png(file = "histogram.png")
8
9 # Create the histogram
10
11 hist(v,xlab = "weight",col = "yellow",border = "blue")
12
13 # Save the file
14
15 dev.off()
16
17
```

The console pane at the bottom shows the execution of the code:

```
> v <- c(8,18,28,7,42,28,18,56,38,43,18)
> png(file = "histogram.png")
> hist(v,xlab = "weight",col = "yellow",border = "blue")
> dev.off()
null device
1
```

Image showing code to generate histogram

Line graphs using R:

A line chart is a graph that connects a series of points by drawing line segments between them. These points are ordered in one of their coordinate (usually the x-coordinate) value. This type of chart is usually used in identifying trends in data.

Syntax:

```
plot(v,type,col,xlab,ylab)
```

`v` - vector containing numerical values

`type` - takes the value "p" to draw only the points, "|" to draw only the lines and "o" to draw both points and lines.

`xlab` - is the label for x axis.

`ylab` - is the label for y axis.

`main` - is the Title of the chart.

`col` - is used to give colors to both the points and lines.

Example:

```
# creating the data for the chart
```

```
v <- c(9,15,22,4,83)
```

```
# Providing a name for the chart file
```

```
png(file = "line_chart.jpg")
```

```
# Plot the chart
```

```
plot(v,type = "o")
```

```
# Save the file.
```

```
dev.off()
```

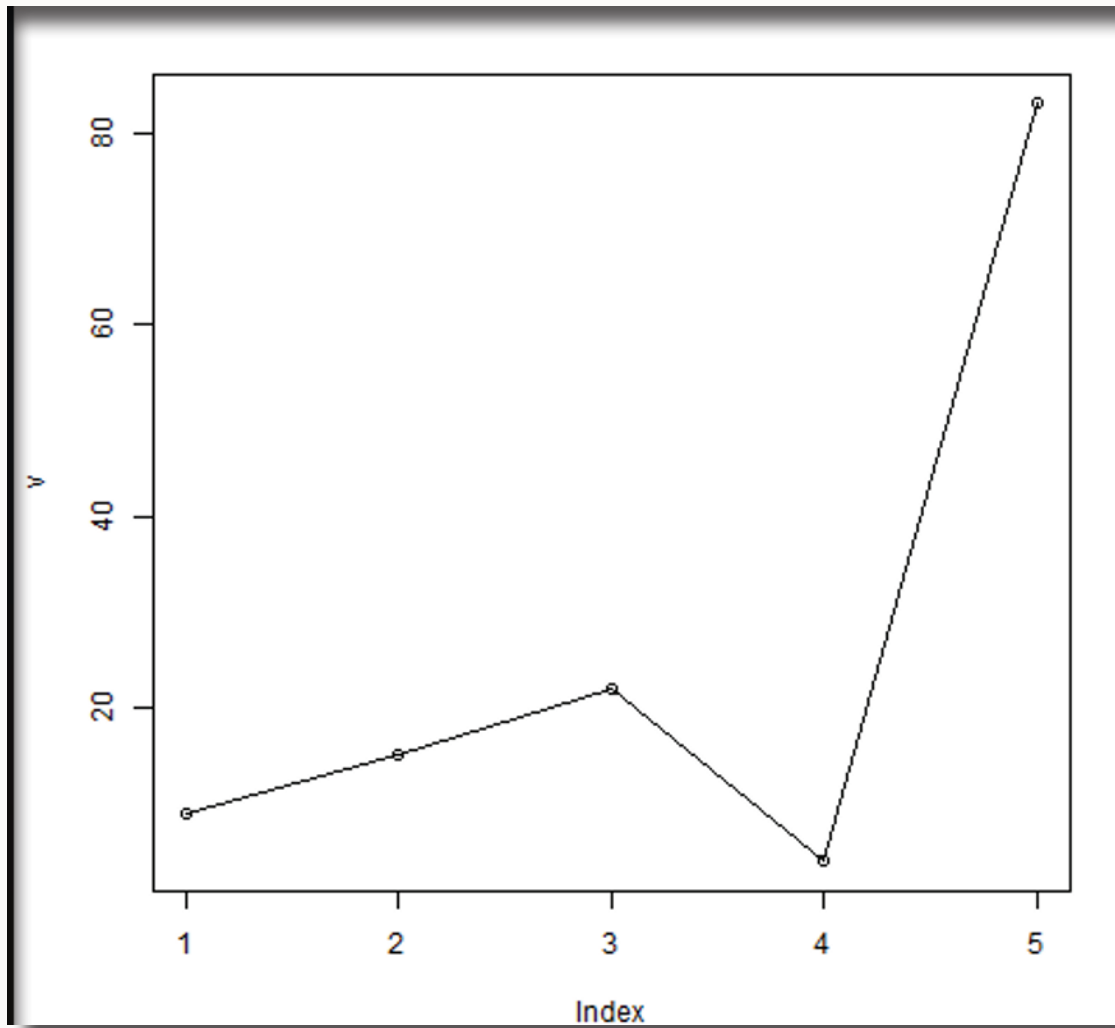
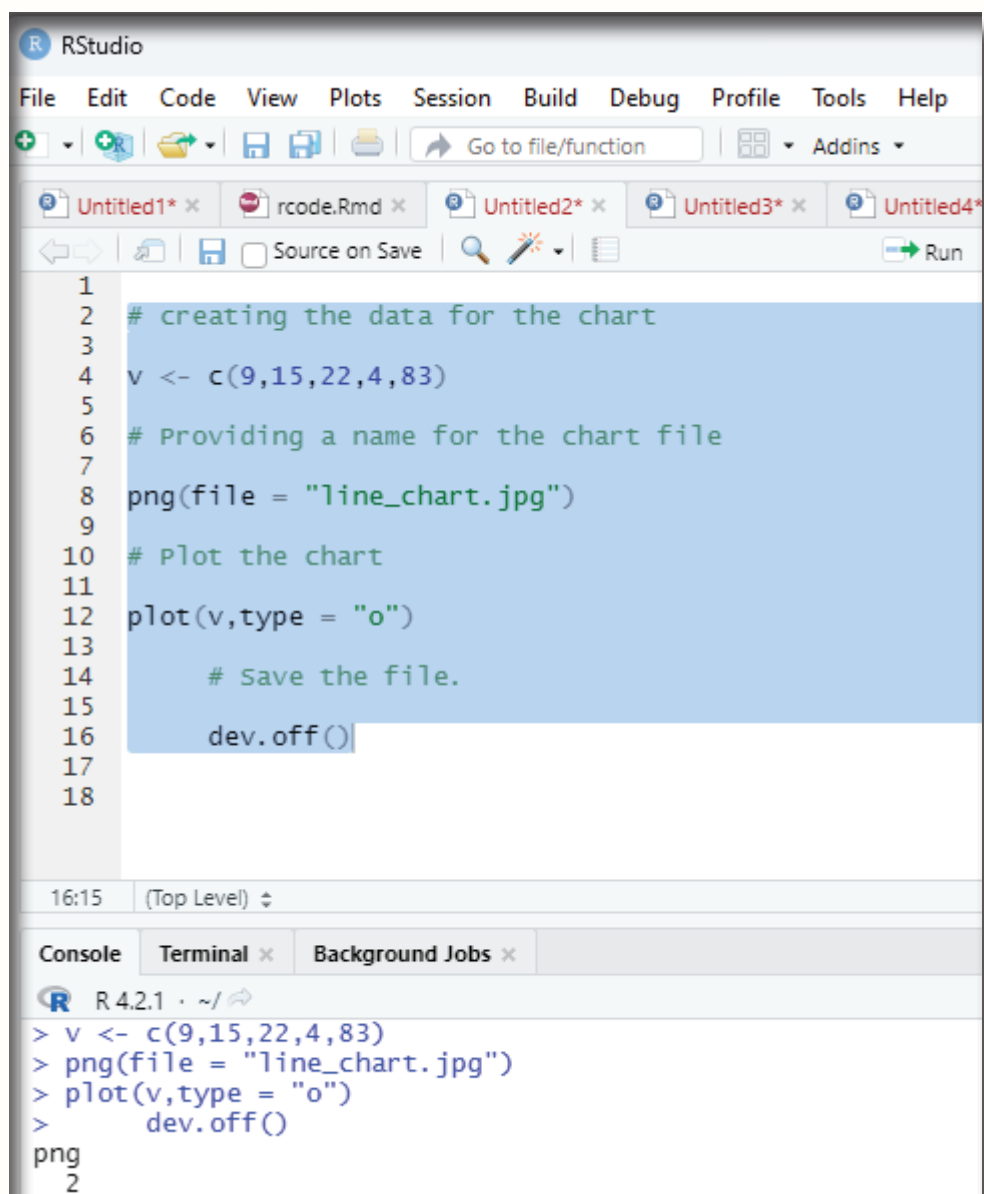


Image showing line graph generated

The image shows the RStudio interface. The top menu bar includes File, Edit, Code, View, Plots, Session, Build, Debug, Profile, Tools, and Help. Below the menu is a toolbar with icons for file operations and a search bar. The editor pane shows a script with the following code:

```
1  
2 # creating the data for the chart  
3  
4 v <- c(9,15,22,4,83)  
5  
6 # Providing a name for the chart file  
7  
8 png(file = "line_chart.jpg")  
9  
10 # Plot the chart  
11  
12 plot(v,type = "o")  
13  
14     # save the file.  
15  
16     dev.off()  
17  
18
```

The console pane at the bottom shows the execution of the code:

```
R 4.2.1 ~/  
> v <- c(9,15,22,4,83)  
> png(file = "line_chart.jpg")  
> plot(v,type = "o")  
>     dev.off()  
png  
2
```

Image showing the code to generate line graph

Line chart with Title, color and labels:

The features of the line chart can be extended using additional parameters. Colors can be added to the points, line etc.

Example:

```
# Create the data for the chart
```

```
v <- c(9,22,45,3,83)
```

```
# Give the chart file a name
```

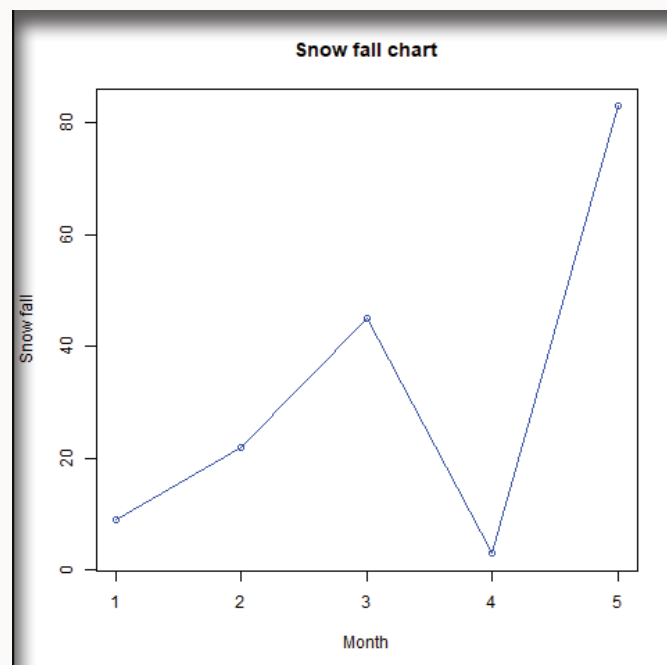
```
png(file = "line_chart_colored.jpg")
```

```
# plot the line graph
```

```
plot(v,type = "o", col = "yellow", xlab = "Month", ylab = "Snow fall",  
      main = "Snow fall chart")
```

```
# Save the file
```

```
dev.off()
```



Line chart (colored)

Multiple lines in a line chart:

More than one line can be drawn on the same chart using the lines() function.

Example:

```
# Create the data for the chart.
```

```
v <- c(7,12,28,3,41)
```

```
t <- c(14,7,6,19,3)
```

```
# Give the chart file a name.
```

```
png(file = "line_chart_2_lines.jpg")
```

```
# Plot the bar chart.
```

```
plot(v,type = "o",col = "red", xlab = "Month", ylab = "Rain fall",  
main = "Rain fall chart")
```

```
lines(t, type = "o", col = "blue")
```

```
# Save the file.
```

```
dev.off()
```

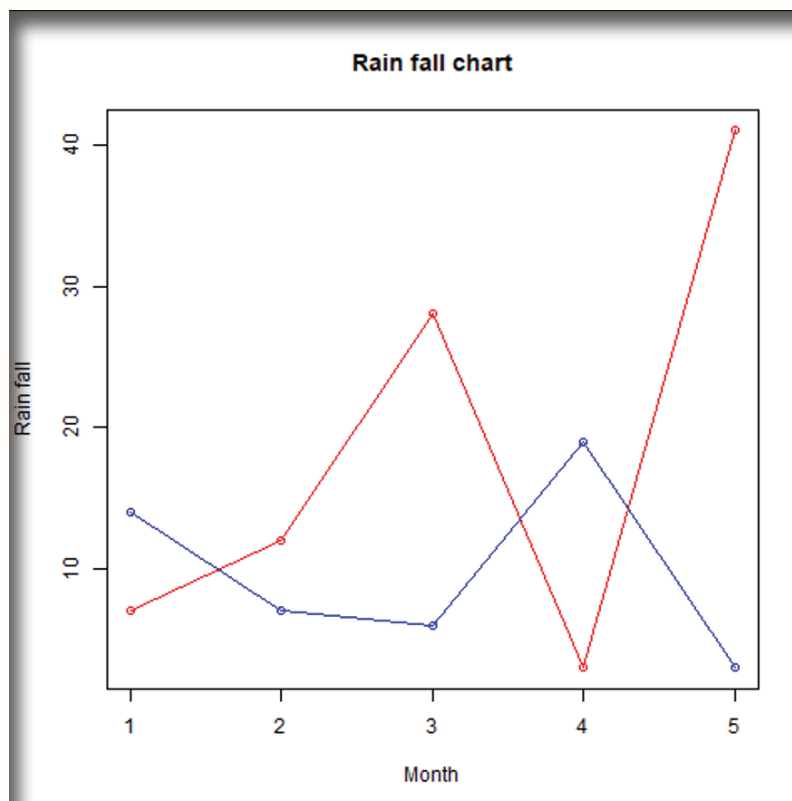
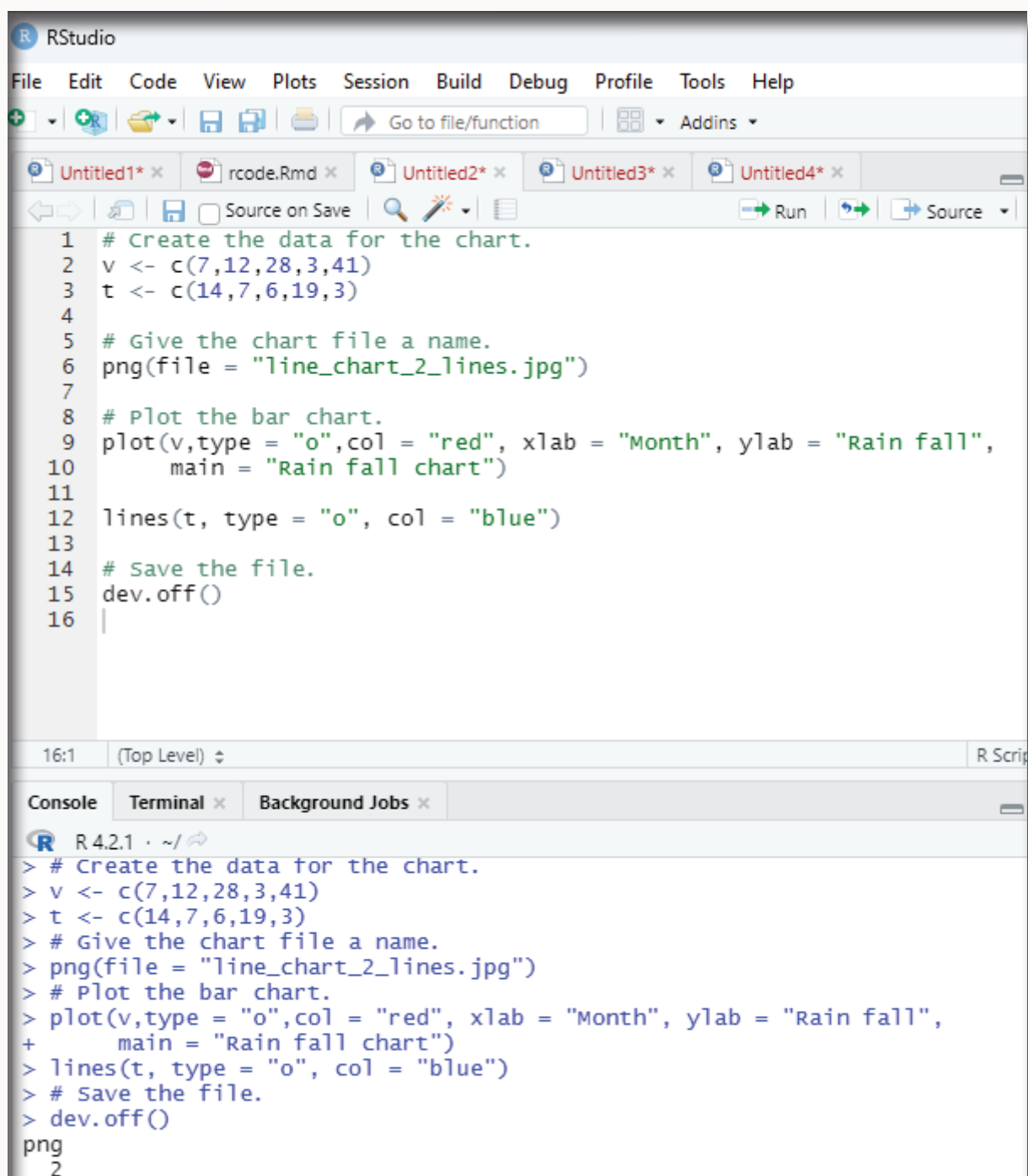


Image showing two line graph



The image shows the RStudio interface with a script editor containing R code to create a line chart. The code defines two vectors, 'v' and 't', and uses 'plot()' and 'lines()' to create a chart with two lines. The console shows the execution of the code.

```
1 # Create the data for the chart.
2 v <- c(7,12,28,3,41)
3 t <- c(14,7,6,19,3)
4
5 # Give the chart file a name.
6 png(file = "line_chart_2_lines.jpg")
7
8 # Plot the bar chart.
9 plot(v,type = "o",col = "red", xlab = "Month", ylab = "Rain fall",
10      main = "Rain fall chart")
11
12 lines(t, type = "o", col = "blue")
13
14 # save the file.
15 dev.off()
16
```

Console output:

```
R 4.2.1 ~/  
> # Create the data for the chart.  
> v <- c(7,12,28,3,41)  
> t <- c(14,7,6,19,3)  
> # Give the chart file a name.  
> png(file = "line_chart_2_lines.jpg")  
> # Plot the bar chart.  
> plot(v,type = "o",col = "red", xlab = "Month", ylab = "Rain fall",  
+      main = "Rain fall chart")  
> lines(t, type = "o", col = "blue")  
> # save the file.  
> dev.off()  
png  
2
```

Image showing code used to generate two line chart

R Scatterplots:

Scatterplots show many points plotted in the cartesian plane. Each point represents the values of two variables. One variable is chosen in the horizontal axis and the other in vertical axis.

Syntax:

```
plot(x, y, main, xlab, ylab, xlim, ylim, axes)
```

x - is the dataset whose values are the horizontal co-ordinates

y - is the dataset whose values are the vertical co-ordinates

main - is the title of the graph

xlab - is the label in the horizontal axis

ylab - is the label in the vertical axis

xlim - is the limits of the values of x used for plotting

ylim - is the limits of the values of y used for plotting

axes - indicates whether both axes should be drawn on the plot

Example:

The dataset "mtcars" is used. The columns "wt" and "mpg" are used for this purpose.

```
data(mtcars)
```

```
input <- mtcars[,c('wt','mpg')]  
print(head(input))
```

On executing the code the following will be displayed:

	wt	mpg
Mazda RX4	2.620	21.0
Mazda RX4 Wag	2.875	21.0
Datsun 710	2.320	22.8
Hornet 4 Drive	3.215	21.4
Hornet Sportabout	3.440	18.7
Valiant	3.460	18.1

Creating the scatterplot:

The below script will create a scatterplot graph for the relation between wt(weight) and mpg(miles per gallon).

```
# Get the input values.  
input <- mtcars[,c('wt','mpg')]
```

```
# Give the chart file a name.  
png(file = "scatterplot.png")
```

```
# Plot the chart for cars with weight between 2.5 to 5 and mileage between 15 and 30.  
plot(x = input$wt,y = input$mpg,  
      xlab = "Weight",  
      ylab = "Milage",  
      xlim = c(2.5,5),  
      ylim = c(15,30),  
      main = "Weight vs Milage"  
)
```

```
# Save the file.  
dev.off()
```

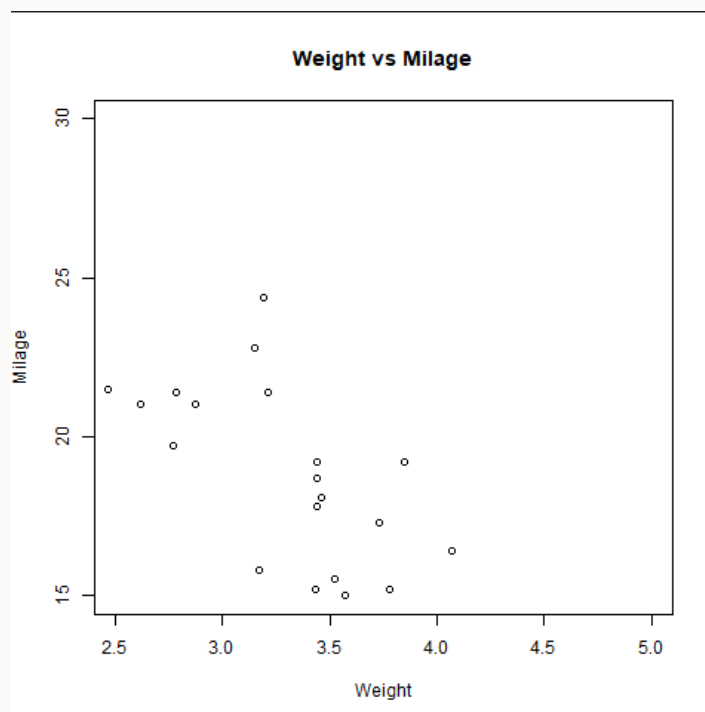


Image showing scatterplot

Scatterplot Matrices:

When there are two variables and the user desires to find the correlation between one variable versus the remaining ones scatterplot can be used. `pairs()` function to create matrices of scatterplots.

Syntax:

`pairs(formula, data)`

formula - represents the series of variable used in pairs.

data - represents the data set from which the variables will be taken.

Example:

```
# Give the chart file a name.  
png(file = "scatterplot_matrices.png")  
  
# Plot the matrices between 4 variables giving 12 plots.  
  
# One variable with 3 others and total 4 variables.  
  
pairs(~wt+mpg+disp+cyl,data = mtcars,  
main = "Scatterplot Matrix")  
  
# Save the file.  
dev.off()
```

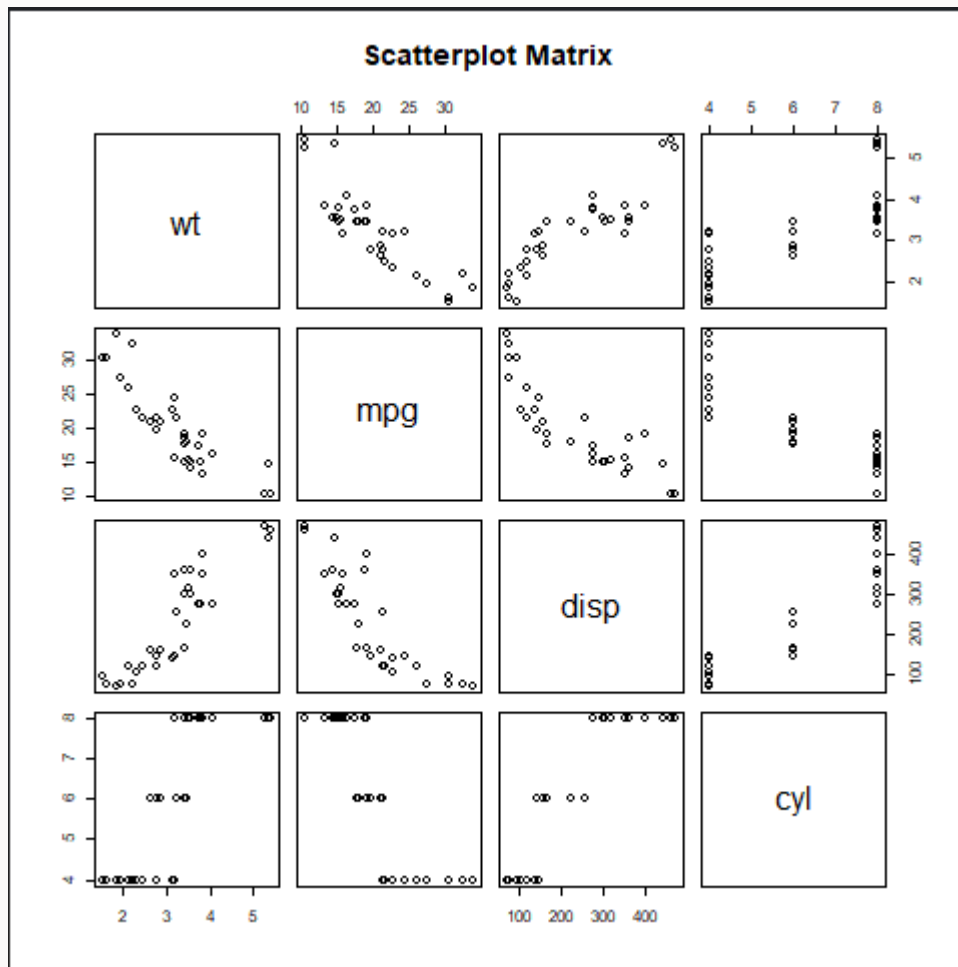


Image showing scatterplot of a matrix

